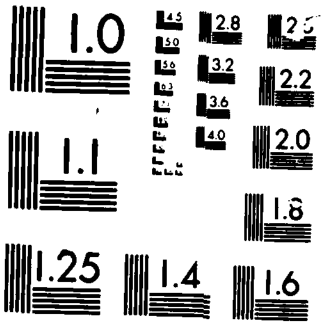
AD-A166 351   ADA (TRADE NAME) TRAINING CURRICULUN REAL-TIME SYSTEMS   1/5
IN ADA L401 TEACHER'S GUIDE VOLUME 1(U) SOFTECH INC
WALTHAM MA 1986 DAAB07-83-C-K514

UNCLASSIFIED   F/G 5/9   NL



MICROCOPY RESOLUTION TEST CHART

1986

Ada® Training Curriculum

Real-Time Systems In Ada
L401
Teacher's Guide
Volume I

Superseded AD-A146782

U.S. Army Communications-Electronics Command
(CECOM)

Contract DAAB07-83-C-K5~~06~~14

Prepared By:

SOFTECH, INC.
460 Totten Pond Road
Waltham, MA 02154

•Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

*Approved For Public Release/Distribution Unlimited

L401

REAL-TIME SYSTEMS IN ADA

L401 -- REAL-TIME SYSTEMS IN ADA
COURSE OUTLINE :

PART 1 ~~I~~, CONCURRENT PROGRAMMING CONCEPTS —
SECTION 1. CONCURRENT PROCESSES,
SECTION 2. REASONS FOR CONCURRENCY,
SECTION 3. CONCURRENT PROGRAMMING PROBLEMS;

PART 2 ~~II~~) ADA TASKING CONCEPTS —
SECTION 4. TASK TYPES AND TASK OBJECTS,
SECTION 5. TASK DECLARATIONS AND TASK TYPES,
SECTION 6. TASK ACTIVATION AND TERMINATION; and

PART 3 ~~III~~. TASK COOPERATION —
SECTION 7) SIMPLE RENDEZVOUS,
SECTION 8) SELECTIVE WAITS,
SECTION 9) SELECT STATEMENTS FOR MAKING ENTRY CALLS,
SECTION 10) AVOIDING DEADLOCK. —

PART 4 ~~IV~~) FUNDAMENTAL TASK DESIGNS —
SECTION 11) SERVER AND USER TASKS,
SECTION 12) MONITORS,
SECTION 13) MESSAGE BUFFERS,
SECTION 14) CYCLIC PROCESSING,
SECTION 15) STREAM-ORIENTED TASK DESIGN;

[illegible] compute programs, [illegible] monitors, [illegible] data processing.

PART 5 ~~V~~) OTHER TASKING FEATURES —
SECTION 16) ABORTING TASKS,
SECTION 17) EXCEPTIONS IN MULTITASK PROGRAMS,
SECTION 18) INTERRUPT ENTRIES,
SECTION 19) ENTRY FAMILIES,
SECTION 20) TASK PRIORITIES;

PART 6 ~~VI~~) IMPROVING PERFORMANCE —
SECTION 21) WHEN AND WHY TO TUNE,
SECTION 22) SHARED VARIABLES,
SECTION 23) MINIMIZING BLOCKING,
SECTION 24) MERGING TASKS,
SECTION 25) NON-CONCURRENT TUNING,
SECTION 26) WHAT'S BEST LEFT TO THE COMPILER; and

PART 7 ~~VII~~) MODULE WRAP-UP —
SECTION 27) A COMPLETE EXAMPLE.

PART I

CONCURRENT PROGRAMMING CONCEPTS

1. CONCURRENT PROCESSES

2. REASONS FOR CONCURRENCY

3. CONCURRENT PROGRAMMING PROBLEMS

INSTRUCTOR NOTES

ALLOW 60 MINUTES FOR THIS SECTION.

THE DISCUSSION OF CONCURRENT PROCESSES IN THIS SECTION SHOULD BE KEPT ALMOST ENTIRELY LANGUAGE-INDEPENDENT.

RECOMMEND THAT STUDENTS READ EXERCISE 1.1 IN THE REAL-TIME ADA WORKBOOK.

Section 1

CONCURRENT PROCESSES

INSTRUCTOR NOTES

A PROCESS IS NOT SYNONYMOUS WITH AN ADA TASK. THE NOTION OF A PROCESS IS INDEPENDENT OF ANY PROGRAMMING LANGUAGE. IN SECTION 4, AN ADA TASK OBJECT WILL BE DESCRIBED AS CONSISTING OF A PROCESS, AN EXCLUSIVE SET OF DATA, AND AN INTERFACE FOR COMMUNICATION WITH OTHER TASKS.

BULLET 3: "CONVENTIONAL PROGRAMS" INCLUDE MOST COMMERCIAL PROGRAMS, ALL PROGRAMS WRITTEN IN STANDARD FORTRAN, AND ALMOST ALL PROGRAMS SHOWN IN MODULES L202 AND L305.

MULTIPLE PROCESSES

- A PROCESS IS A SEQUENCE OF ACTIONS PERFORMED IN CARRYING OUT A PROGRAM.

- SEVERAL PROCESSES CAN BE IN PROGRESS AT THE SAME TIME.

- "CONVENTIONAL" PROGRAMS ARE WRITTEN FOR A SINGLE PROCESS.

  -- THEY SPECIFY ONE SEQUENCE OF ACTIONS TO BE PERFORMED IN A SPECIFIC ORDER.

- IN Ada, YOU CAN WRITE MULTI-PROCESS PROGRAMS.

  -- THEY SPECIFY TWO OR MORE SEQUENCES OF ACTIONS THAT MAY BE IN PROGRESS AT THE SAME TIME.

- MODULE L401 CONCENTRATES ON MULTI-PROCESS PROGRAMS.

INSTRUCTOR NOTES

THIS SLIDE MODELS PROGRAMS WITH RECIPES.

THE FIRST "PROGRAM" DESCRIBES EIGHT ACTIONS TO BE PERFORMED IN SEQUENCE TO COOK MACARONI.

THE SECOND "PROGRAM" DESCRIBES TWENTY ACTIONS TO BE PERFORMED TO COOK MACARONI AND CREAM SAUCE. THE MACARONI AND THE CREAM SAUCE CAN BE PREPARED AT THE SAME TIME, SO THIS "PROGRAM" CONSISTS OF TWO PROCESSES. THE ACTIONS LISTED FOR A GIVEN PROCESS MUST BE PERFORMED IN SEQUENCE.

SINGLE- AND MULTIPLE-PROCESS PROGRAMS

- EXAMPLE OF A SINGLE-PROCESS PROGRAM (TO COOK MACARONI):

1. FILL POT WITH WATER.
2. TURN ON HIGH FLAME.
3. WAIT UNTIL WATER IS BOILING.
4. ADD MACARONI TO POT.
5. LOWER FLAME.
6. WAIT UNTIL MACARONI IS COOKED.
7. TURN OFF FLAME.
8. EMPTY CONTENTS OF POT INTO COLANDER.

- EXAMPLE OF A TWO-PROCESS PROGRAM (TO COOK MACARONI AND CREAM SAUCE):

PROCESS 1 (TO COOK MACARONI):

1. FILL POT WITH WATER.
2. TURN ON HIGH FLAME.
3. WAIT UNTIL WATER IS BOILING.
4. ADD MACARONI TO POT.
5. LOWER FLAME.
6. WAIT UNTIL MACARONI IS COOKED.
7. TURN OFF FLAME.
8. EMPTY CONTENTS OF POT INTO COLANDER.

PROCESS 2 (TO COOK CREAM SAUCE):

1. PLACE BUTTER IN SAUCEPAN.
2. TURN ON LOW FLAME.
3. WAIT UNTIL BUTTER IS MELTED.
4. ADD FLOUR.
5. MIX WELL.
6. ADD MILK.
7. MIX WELL.
8. RAISE FLAME TO MEDIUM.
9. WAIT UNTIL MIXTURE BOILS, STIRRING OCCASIONALLY.
10. LOWER FLAME.
11. STIR CONSTANTLY FOR TWO MINUTES.
12. TURN OFF FLAME.

INSTRUCTOR NOTES

OVERLAPPED CONCURRENCY AND INTERLEAVED CONCURRENCY ARE DEFINED ON THIS SLIDE AND ILLUSTRATED ON THE TWO FOLLOWING SLIDES.

THE NOTION OF VIRTUAL PROCESSORS IS INTRODUCED AFTERWARD. POSTPONE DISCUSSION OF THAT NOTION FOR NOW.

PROCESSES AND PROCESSORS

- A PROCESSOR IS AN AGENT (SUCH AS A CPU) THAT PERFORMS ACTIONS IN SEQUENCE.

- DIFFERENT PROCESSES CAN BE PERFORMED SIMULTANEOUSLY BY DIFFERENT PROCESSORS.

  -- THIS IS CALLED OVERLAPPED CONCURRENCY.

- ALTERNATIVELY, A SINGLE PROCESSOR CAN TAKE TURNS ATTENDING TO SEVERAL PROCESSES AND PERFORMING A FEW ACTIONS *FROM EACH PROCESS IN TURN*.

  -- THIS IS CALLED INTERLEAVED CONCURRENCY.

  -- EACH PROCESS PROGRESSES WHEN ITS TURN COMES.

INSTRUCTOR NOTES

EACH STRAIGHT HORIZONTAL LINE CONTAINS THE SEQUENCE OF ACTIONS CONSTITUTING A SINGLE PROCESS. EACH SOLID PATH CONTAINS THE SEQUENCE OF ACTIONS PERFORMED BY A SINGLE PROCESSOR. IN THE ILLUSTRATION OF OVERLAPPED CONCURRENCY, THESE ARE IDENTICAL.

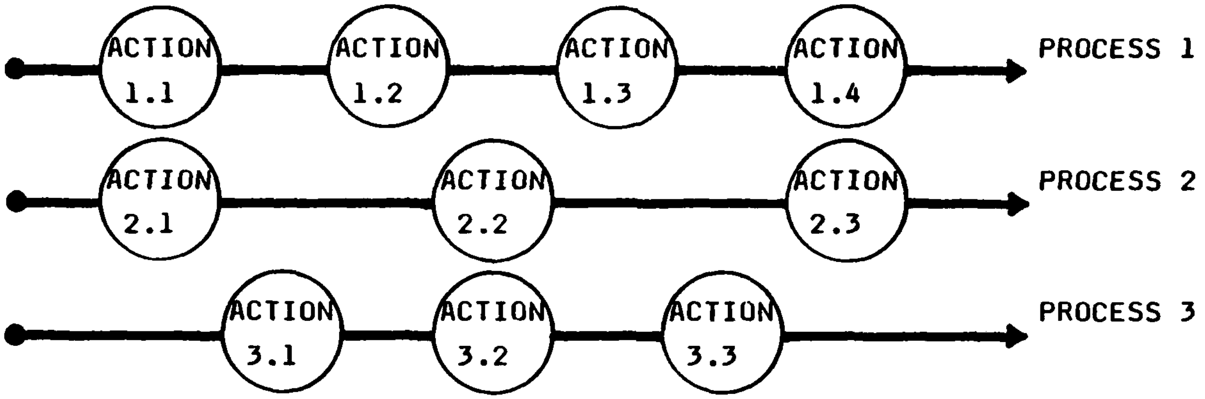
MAKE THE FOLLOWING OBSERVATIONS:

IN INTERLEAVED CONCURRENCY, WHEN ONE PROCESSOR IS BEING SHARED BY SEVERAL PROCESSES, THE PROCESSES PROGRESS MORE SLOWLY.

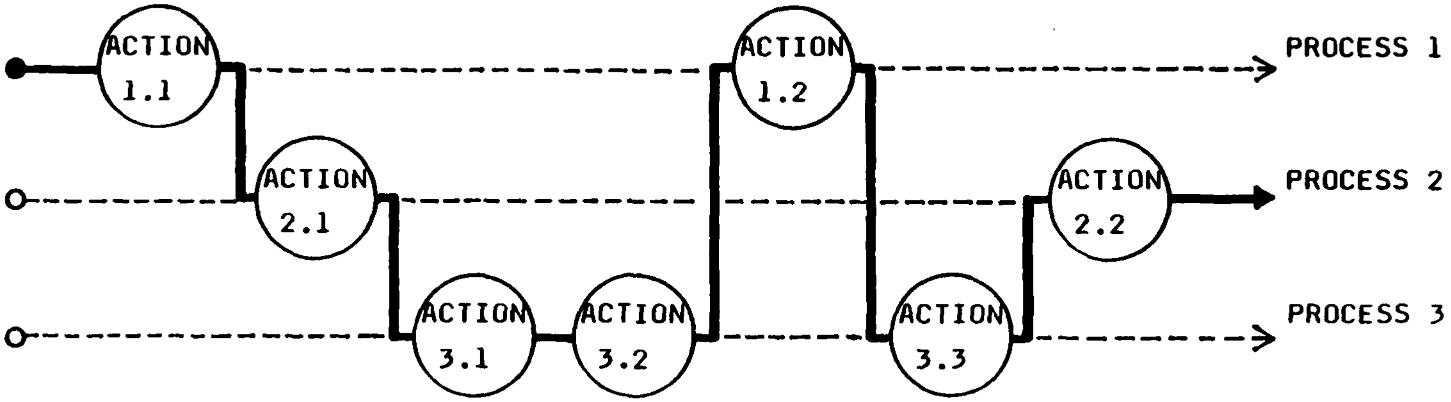
ACTIONS DO NOT OCCUR WITHIN THE SAME ORDER IN BOTH ILLUSTRATIONS (E.G., 1.2 AND 3.2; 2.2 AND 3.3). WITHIN A GIVEN PROCESS, HOWEVER, ACTIONS OCCUR IN RIGID SEQUENCE. (JUST POINT THIS OUT. DETAILED DISCUSSION OF ASYNCHRONISM FOLLOWS LATER.)

OVERLAPPED AND INTERLEAVED CONCURRENCY

- OVERLAPPED CONCURRENCY:



- INTERLEAVED CONCURRENCY:



INSTRUCTOR NOTES

THE CHEFS ARE THE PROCESSORS, AND THE PROCESSES ARE THE PREPARATION OF MACARONI AND PREPARATION OF CREAM SAUCE. (SEE THE BOTTOM HALF OF SLIDE 1-2.)

OVERLAPPED AND INTERLEAVED CONCURRENCY -- EXAMPLE

- OVERLAPPED CONCURRENCY:

  TWO CHEFS, ONE COOKING MACARONI WHILE THE OTHER COOKS CREAM SAUCE.

- INTERLEAVED CONCURRENCY:

  ONE CHEF COOKING BOTH MACARONI AND CREAM SAUCE, SWITCHING HIS ATTENTION BACK AND FORTH BETWEEN THE TWO.

- EITHER WAY, THE SET OF DIRECTIONS FOR MACARONI IS CARRIED OUT IN SEQUENCE AND THE SET OF DIRECTIONS FOR CREAM SAUCE IS CARRIED OUT IN SEQUENCE.

INSTRUCTOR NOTES

HERE, ONE CHEF IS COOKING BOTH THE MACARONI AND THE CREAM SAUCE, AS DESCRIBED IN BULLET 2 OF THE PREVIOUS SLIDE.

WHILE ATTENDING TO THE MACARONI, THE CHEF WEARS HIS CHEF JEKYLL HAT. WHEN HE SWITCHES HIS ATTENTION TO THE CREAM SAUCE, HE DONS HIS CHEF HYDE HAT. JUST AS ROBERT LOUIS STEVENSON'S DR. JEKYLL WAS ONE PERSON WHO CREATED THE ILLUSION OF BEING TWO PEOPLE, SO THE CHEF CREATES THE ILLUSION THAT THERE ARE TWO CHEFS -- A CHEF JEKYLL WHO DEVOTES HIS ATTENTION TO MACARONI AND A CHEF HYDE WHO DEVOTES HIS ATTENTION TO CREAM SAUCE.

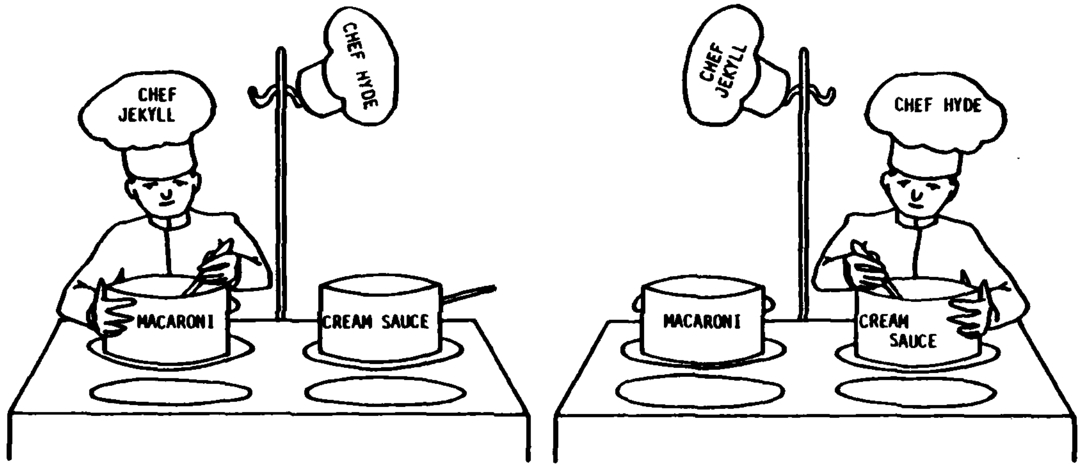
IN COMPUTER SYSTEMS, A PROCESSOR THAT TAKES TURNS EXECUTING DIFFERENT PROCESSES CREATES THE ILLUSION OF SEVERAL SLOWER PROCESSORS EACH DEVOTED TO A SINGLE PROCESS. (IT MAY BE USEFUL TO SHOW SLIDE 1-4 AGAIN TO ILLUSTRATE THIS POINT.) EACH OF THESE ILLUSORY PROCESSORS IS CALLED A VIRTUAL PROCESSOR.

(BULLET 3 SAYS "MAY CORRESPOND" RATHER THAN "CORRESPONDS" BECAUSE OVERLAPPED CONCURRENCY MAY CONSIST OF, SAY, THREE PROCESSORS TAKING TURNS PERFORMING TEN PROCESSES. PHYSICALLY, THREE PROCESSES ARE OVERLAPPED AT ANY MOMENT. LOGICALLY, THERE ARE TEN VIRTUAL PROCESSORS.)

THINKING IN TERMS OF VIRTUAL PROCESSORS ALLOWS A PROGRAMMER TO APPROACH A PROBLEM FROM A HIGHER LEVEL OF ABSTRACTION. THE DISTINCTION BETWEEN OVERLAPPED AND INTERLEAVED CONCURRENCY THEN BECOMES AN IMPLEMENTATION DETAIL.

VIRTUAL PROCESSORS

- INTERLEAVED CONCURRENCY CREATES THE ILLUSION THAT THERE ARE A NUMBER OF SLOWER PROCESSORS, EACH PERFORMING ONE OF THE PROCESSES.



- THE PROCESSORS THAT APPEAR TO EXIST ARE CALLED VIRTUAL PROCESSORS.

- WITH OVERLAPPED CONCURRENCY, EACH VIRTUAL PROCESSOR MAY CORRESPOND TO AN ACTUAL PROCESSOR.

- A PROGRAMMER CAN THINK IN TERMS OF VIRTUAL PROCESSORS WITHOUT WORRYING ABOUT WHETHER CONCURRENCY IS INTERLEAVED OR OVERLAPPED.

INSTRUCTOR NOTES

AN ADA PROGRAM MAY RUN UNDER SOME OPERATING SYSTEM OR ON A BARE MACHINE.

WHEN IT RUNS UNDER AN OPERATING SYSTEM, THE OPERATING SYSTEM PROVIDES MOST OR ALL OF THE SERVICES THAT THE RUNTIME SYSTEM MAKES AVAILABLE TO THE PROGRAM.

WHEN IT RUNS ON A BARE MACHINE, THE RUNTIME SYSTEM ITSELF IMPLEMENTS THESE SERVICES.

BULLET 3 PROVIDES ONLY A PARTIAL LIST OF THE SERVICES PROVIDED BY A RUNTIME SYSTEM. IN L401, WE ARE INTERESTED PRIMARILY IN THE THIRD ITEM LISTED, VIRTUAL PROCESSORS.

THE RUNTIME SYSTEM

- AN ADA PROGRAM RUNS IN AN ENVIRONMENT THAT INCLUDES A RUNTIME SYSTEM.

- A RUNTIME SYSTEM IS THE INTERFACE BETWEEN A RUNNING ADA PROGRAM AND THE UNDERLYING MACHINE OR OPERATING SYSTEM.

- THE RUNTIME SYSTEM PROVIDES SERVICES NEEDED BY THE RUNNING ADA PROGRAM, SUCH AS:

  -- INPUT AND OUTPUT OPERATIONS

  -- ALLOCATION AND DEALLOCATION OF STORAGE

  -- VIRTUAL PROCESSORS TO RUN MULTIPLE PROCESSES

- THE RUNTIME SYSTEM IS "THE EXECUTIVE."

INSTRUCTOR NOTES

BULLET 1: ACHIEVING INTERLEAVING:

AN IN-DEPTH DISCUSSION OF PROCESSOR SCHEDULING IS BEYOND THE SCOPE OF THIS MODULE, BUT HERE ARE A FEW BRIEF DEFINITIONS:

ROUND-ROBIN TIME-SLICING: THE PROCESSOR EXECUTES EACH PROCESS FOR A SPECIFIED AMOUNT OF TIME, THEN SWITCHES TO THE NEXT PROCESS THAT IS WAITING TO EXECUTE. PROCESSES TAKE TURNS USING THE PROCESSOR IN A FIXED ORDER.

TIME-SLICING WITH PRIORITIES: THE PROCESSOR EXECUTES EACH PROCESS FOR A SPECIFIED AMOUNT OF TIME, THEN SWITCHES TO THE HIGHEST-PRIORITY PROCESS THAT IS WAITING TO EXECUTE.

PRE-EMPTIVE SCHEDULING: A PROCESS EXECUTES UNTIL AN EXTERNAL EVENT OCCURS, SUCH AS AN INTERRUPT OR SOME ACTION THAT A HIGHER-PRIORITY PROCESS WAS WAITING FOR.

VOLUNTARY SUSPENSION: A PROCESS CONTINUES TO EXECUTE UNTIL IT MUST WAIT FOR ANOTHER PROCESS TO DO SOMETHING, THEN RELINQUISHES THE PROCESSOR.

THESE ARE ONLY A FEW OF THE MANY POSSIBLE WAYS OF ACHIEVING INTERLEAVING.

BULLETS 2-4:

THE RULES OF ADA REQUIRE THE RUNTIME SYSTEM TO PROVIDE VIRTUAL PROCESSORS, BUT DO NOT CONSTRAIN HOW THIS IS DONE.

THE RULES OF ADA CAN BE UNDERSTOOD IN TERMS OF VIRTUAL PROCESSORS WITHOUT UNDERSTANDING THE UNDERLYING IMPLEMENTATION.

ALTERNATIVE RUNTIME SYSTEMS

- THERE ARE MANY DIFFERENT WAYS TO BUILD RUNTIME SYSTEMS.
  - ALTERNATIVE WAYS TO PROVIDE VIRTUAL PROCESSORS
    - OVERLAPPED CONCURRENCY WITH EACH VIRTUAL PROCESSOR CORRESPONDING TO A PHYSICAL PROCESSOR.
    - INTERLEAVED CONCURRENCY ON A SINGLE PHYSICAL PROCESSOR.
    - A COMBINATION, E.G., INTERLEAVING FIVE PROCESSES ON TWO OVERLAPPED PROCESSORS.
  - ALTERNATIVE WAYS TO ACHIEVE INTERLEAVING, INCLUDING:
    - ROUND-ROBIN TIME-SLICING
    - TIME-SLICING WITH PRIORITIES
    - PRE-EMPTIVE SCHEDULING
    - VOLUNTARY SUSPENSION
  - ETC., ETC., ETC.
- THE RULES OF ADA CAN BE MET BY ANY OF THESE CHOICES.
- THE INNER WORKINGS OF THE RUNTIME SYSTEM ARE HIDDEN FROM THE PROGRAMMER.
- AN ADA PROGRAM CAN USUALLY BE WRITTEN SO THAT ITS LOGIC DOES NOT DEPEND ON THE INNER WORKINGS OF THE RUNTIME SYSTEM.

INSTRUCTOR NOTES

BULLET 1:

A CORRECT PROGRAM MAY PRODUCE DIFFERENT RESULTS WITH DIFFERENT RUNTIME SYSTEMS. BY "VALID" WE MEAN THAT ANY OF THESE RESULTS ARE ACCEPTABLE.

(THE FIRST BULLET ASSUMES THAT ALL THE RUNTIME SYSTEMS FOR A GIVEN COMPILER HAVE THE SAME INTERFACE IN TERMS OF ADA. FOR EXAMPLE, THE MEANING OF LOW-LEVEL FEATURES IS ASSUMED TO BE THE SAME, NO MATTER WHICH RUNTIME SYSTEM THE COMPILER IS WORKING WITH. THUS CHANGING A RUNTIME SYSTEM SHOULD NOT NECESSITATE CHANGING ADA SOURCE TEXT, EXCEPT PERHAPS TO IMPROVE PERFORMANCE.)

BULLET 4:

CUSTOMIZING OR REWRITING AN ADA RUNTIME SYSTEM IS EQUIVALENT TO WRITING AN EXECUTIVE AS THE FIRST STEP IN IMPLEMENTING A REAL-TIME SYSTEM.

DIFFERENT RUNTIME SYSTEMS ARE BETTER FOR DIFFERENT APPLICATIONS

- ADA PROGRAMS CAN BE WRITTEN TO PRODUCE EQUALLY VALID ANSWERS WITH ANY RUNTIME SYSTEM.

- THE CHOICE OF A RUNTIME SYSTEM MAY PROFOUNDLY AFFECT THE PERFORMANCE OF THE PROGRAM.

- DIFFERENT RUNTIME SYSTEMS WITH DIFFERENT PERFORMANCE CHARACTERISTICS MAY BE AVAILABLE FOR THE SAME TARGET MACHINE.

- FOR SOME PROJECTS IT MAY BE NECESSARY TO CUSTOM-BUILD A RUNTIME SYSTEM, OR TO MODIFY AN EXISTING ONE.

- THE GOOD NEWS:

  - THIS MAKES IT EASIER TO TAILOR SYSTEMS WRITTEN IN ADA TO PROJECT REQUIREMENTS.

  - AS TIME GOES ON, MORE AND MORE RUNTIME SYSTEMS WILL BE AVAILABLE OFF THE SHELF.

INSTRUCTOR NOTES

THIS SLIDE PROVIDES AN OVERVIEW OF CONCEPTS THAT ARE ADDRESSED IN DETAIL ON THE NEXT FEW SLIDES.

ASYNCHRONOUS PROCESSES

- PROCESSES ARE GENERALLY ASYNCHRONOUS. THAT IS, THEY PROCEED AT DIFFERENT RATES.

- THE RELATIVE PROGRESS OF ONE PROCESS WITH RESPECT TO ANOTHER IS UNPREDICTABLE.

- THIS CAN CREATE PROBLEMS WHEN ONE PROCESS TRIES TO USE DATA BEING PRODUCED BY ANOTHER.

- THERE ARE WAYS TO CAUSE PROCESSES TO SYNCHRONIZE MOMENTARILY.

INSTRUCTOR NOTES

- BULLET 2:

  -- ITEM 2: FOR EXAMPLE, PROCESSES MAY HAVE DIFFERENT PRIORITIES

- BULLET 3:

EXTERNAL INPUTS CAN MAKE IT IMPOSSIBLE IN PRINCIPLE TO PREDICT THE RELATIVE SPEEDS OF DIFFERENT PROCESSES. EVEN IF THIS WERE NOT THE CASE, THE FACTORS INVOLVED ARE SO COMPLEX THAT PREDICTION WOULD BE IMPOSSIBLE IN PRACTICE.

BY REGARDING THE RELATIVE PROGRESS OF DIFFERENT PROCESSES AS ESSENTIALLY RANDOM, WE GREATLY REDUCE THE NUMBER OF DETAILS WITH WHICH WE HAVE TO BE CONCERNED.

THIS "RANDOM FACTOR" IS ABSENT FROM MOST SEQUENTIAL PROGRAMS. ONE OF THE BIGGEST HURDLES A SEQUENTIAL PROGRAMMER FACES WHEN LEARNING CONCURRENT PROGRAMMING IS RECOGNIZING AND COPING WITH NONDETERMINISM.

WHY PROCESSES ARE ASYNCHRONOUS

- THEY MAY BE RUNNING ON PHYSICAL PROCESSORS WITH DIFFERENT SPEEDS.
- THEY MAY BE RUNNING ON VIRTUAL PROCESSORS IMPLEMENTED BY INTERLEAVING.
    - -- THE AMOUNT OF TIME SPENT ON ONE PROCESS BEFORE SWITCHING TO ANOTHER MAY NOT BE UNIFORM.
    - -- SELECTION OF THE NEXT PROCESS TO BE PERFORMED MAY NOT TREAT ALL PROCESSES EQUALLY.
    - -- PROCESSES MAY VOLUNTARILY *SUSPEND THEMSELVES* UNTIL THE OCCURRENCE OF SOME EXTERNAL EVENT:
        - AN INTERRUPT
        - PASSAGE OF A SPECIFIED AMOUNT OF TIME
        - THE PERFORMANCE OF SOME ACTION BY ANOTHER PROCESS
- TIMING DEPENDS NOT ONLY ON THE RUNTIME SYSTEM, BUT ALSO ON THE EXTERNAL INPUTS DURING A GIVEN EXECUTION.

INSTRUCTOR NOTES

- BULLET 2:

  THIS PROBLEM AND OTHERS ARE ADDRESSED IN GREATER DETAIL IN SECTION 3.

  -- ITEM 2: THIS KIND OF OVERWRITING MAY BE ACCEPTABLE IN SOME SYSTEMS, DISASTROUS IN OTHERS.

- BULLET 3:

  THE ASSIGNMENTS SHOWN FOR EACH PROCESS MODEL WHAT TYPICALLY HAPPENS AT THE MACHINE LANGUAGE LEVEL WHEN ASSIGNMENTS LIKE X := X + 2 AND X := 2 * X ARE EXECUTED. THE VARIABLES R1 AND R2 MODEL THE REGISTERS IN WHICH COMPUTATIONS ARE CARRIED OUT. ASSIGNMENTS TO R1 OR R2 MODEL LOAD INSTRUCTIONS AND ASSIGNMENTS FROM R1 OR R2 TO X MODEL STORE INSTRUCTIONS.

  THE "ANSWER" TO THE QUESTION IS GIVEN ON THE NEXT SLIDE. PAUSE BEFORE PROCEEDING TO HAVE STUDENTS FIGURE OUT THE ANSWER (OR LACK THEREOF) THEMSELVES.

SOME IMPLICATIONS OF ASYNCHRONISM

- NOTHING MAY BE ASSUMED ABOUT THE RELATIVE SPEEDS OF PROCESSES.

- IF ONE PROCESS PRODUCES DATA THAT IS USED BY A SECOND PROCESS, SPECIAL MEASURES MAY BE NECESSARY TO ENSURE THAT:

  -- THE USING PROCESS DOES NOT TRY TO USE DATA BEFORE THE PRODUCING PROCESS HAS PRODUCED IT

  -- THE PRODUCING PROCESS DOES NOT PRODUCE NEW DATA, OVERWRITING OLD DATA, BEFORE THE USING PROCESS HAS USED THE OLD DATA

- IF TWO PROCESSES CAN UPDATE THE SAME DATA, THE RESULTS MAY BE UNPREDICTABLE:

```
PROCESS 1                     PROCESS 2

R1 := X;                      R2 := X;
R1 := R1 + 2;                 R2 := 2 * R2;
X := R1;                      X := R2;
```

IF X = 1 BEFORE THE TWO PROCESSES START, WHAT IS THE VALUE OF X WHEN THE PROCESSES COMPLETE?

INSTRUCTOR NOTES

THE VALUE LEFT IN X IS UNPREDICTABLE BECAUSE IT DEPENDS ON THE RELATIVE PROGRESS OF THE TWO PROCESSES.

QUICKLY RUN THROUGH THE FOUR EXAMPLES.

LIKE INTERLEAVINGS (1) AND (2), THE SIXTEEN INTERLEAVINGS NOT SHOWN INVOLVE EXECUTING THE FIRST TWO INSTRUCTIONS OF EACH PROCESS IN SOME ORDER, AND THEN EXECUTING THE TWO ASSIGNMENTS TO X IN SOME ORDER. IN EACH CASE, R1 CONTAINS 3 AND R2 CONTAINS 2 JUST BEFORE THE ASSIGNMENTS TO X, AND WHICHEVER OF THE TWO ASSIGNMENTS OCCURS LATER GIVES X ITS FINAL VALUE.

POSSIBLE OUTCOMES OF ASYNCHRONOUS EXECUTION

①

PROCESS 1	PROCESS 2	X	R1	R2
		1		
R1 := X;			1	
	R2 := X;			1
R1 := R1+2;			3	
	R2 := 2*R2;			2
X := R1;		3		
	X := R2;	②		

②

PROCESS 1	PROCESS 2	X	R1	R2
		1		
	R2 := X;			1
R1 := X;			1	
	R2 := 2*R2;			2
R1 := R1+2;			3	
	X := R2;	2		
X := R1;		③		

③

PROCESS 1	PROCESS 2	X	R1	R2
		1		
R1 := X;			1	
R1 := R1+2;			3	
X := R1;		3		
	R2 := X;			3
	R2 := 2*R2;			6
	X := R2;	⑥		

④

PROCESS 1	PROCESS 2	X	R1	R2
		1		
	R2 := X;			1
	R2 := 2*R2;			2
	X := R2;	2		
R1 := X;			2	
R1 := R1+2;			4	
X := R1;		④		

THERE ARE SIXTEEN OTHER INTERLEAVINGS, BESIDES THE FOUR SHOWN HERE, ALL OF WHICH LEAVE EITHER 2 OR 3 IN X.

INSTRUCTOR NOTES

- BULLET 2:

  -- EXAMPLE 1: SEE PAGE 1-13i.

  -- EXAMPLE 2: THESE SYNCHRONIZATION POINTS REQUIRE ALL OF THE ACTIONS IN PROCESS 1 TO BE EXECUTED BEFORE ANY OF THE ACTIONS IN PROCESS 2. THE RESULT IS PREDICTABLE, BUT THERE IS NO CONCURRENCY.

- BULLET 3:

  BY FORCING A PROCESS THAT GETS AHEAD TO WAIT FOR A PROCESS THAT FALLS BEHIND, SYNCHRONIZATION CAN CAUSE PROCESSES TO MANIFEST THE SAME AVERAGE SPEED IN THE LONG RUN.

- BULLET 4:

  ADA'S MECHANISMS FOR SYNCHRONIZATION ARE EXPLAINED IN PART III OF L401 (SECTIONS 7-10).

SYNCHRONIZATION

- SYNCHRONIZATION IS A WAY TO ASSURE THAT ONE PROCESS IS AT A SPECIFIED POINT AT THE SAME TIME THAT THE OTHER PROCESS IS AT A SPECIFIED POINT. (THESE POINTS ARE CALLED SYNCHRONIZATION POINTS.)

- SYNCHRONIZATION REDUCES THE NUMBER OF POSSIBLE INTERLEAVINGS.

  -- EXAMPLE 1:

```
PROCESS 1                                      PROCESS 2

R1 := X;                                       R2 := X;
R1 := R1 + 2;                                  R2 := 2 * R2;
    ●<- - - - - SYNCHRONIZATION - - - - ->●
X := R1;             POINTS                    X := R2;
```

  X CAN ONLY END UP HOLDING 2 OR 3.

  -- EXAMPLE 2:

```
PROCESS 1                                      PROCESS 2

R1 := X;                SYNCHRONIZATION - - - ->●
R1 := R1 + 2;                                  R2 := X;
X := R1;                     POINTS            R2 := 2 * R2;
    ●<- - - - - -                              X := R2;
```

  X CAN ONLY END UP HOLDING 6.

- SYNCHRONIZATION DOES NOT CONTROL THE RELATIVE SPEEDS OF PROCESSES BETWEEN SYNCHRONIZATION POINTS, BUT FORCES A FAST PROCESS TO WAIT AT A SYNCHRONIZATION POINT WHILE A SLOW ONE CATCHES UP.

- PROCESSES MUST SYNCHRONIZE TO INTERACT IN A PREDICTABLE WAY.

INSTRUCTOR NOTES

"SYNCHRONIZING WITH THE CLOCK" MEANS GUARANTEEING THAT A PROCESS WILL BE AT A CERTAIN POINT WHEN THE CLOCK IS AT A CERTAIN POINT. IT IS DIFFERENT FROM SYNCHRONIZING WITH AN ORDINARY PROCESS, BECAUSE THE CLOCK CAN'T BE MADE TO WAIT WHILE ANOTHER PROCESS CATCHES UP.

CONCURRENT PROGRAMMING AND REAL-TIME PROGRAMMING

- CONCURRENT PROGRAMMING IS THE CONSTRUCTION OF A PROGRAM SPECIFYING ACTIONS FOR MULTIPLE PROCESSES.
  - -- IN CONCURRENT PROGRAMMING, COOPERATING PROCESSES MUST SYNCHRONIZE WITH EACH OTHER.
- REAL-TIME PROGRAMMING IS A SPECIAL FORM OF CONCURRENT PROGRAMMING IN WHICH ACTIONS MUST BE PERFORMED WITHIN SPECIFIED TIME INTERVALS.
  - -- PROCESSES MUST SYNCHRONIZE NOT ONLY WITH EACH OTHER, BUT WITH THE CLOCK.

INSTRUCTOR NOTES

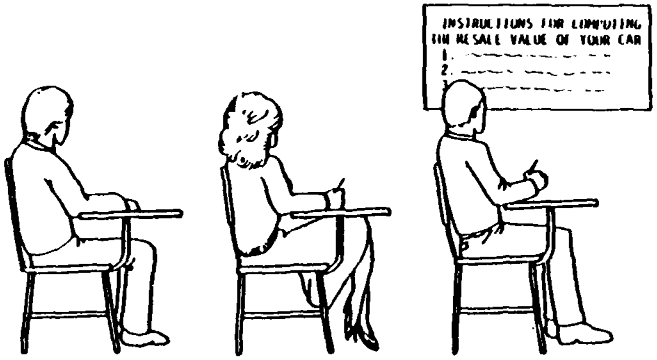
THE THREE PEOPLE FOLLOWING THE SAME INSTRUCTIONS IN THIS PICTURE REPRESENT MULTIPLE PROCESSES EXECUTING THE SAME PROGRAM.

JUST AS ONE PERSON MAY STILL BE ON INSTRUCTION 1 WHILE ANOTHER HAS MOVED ON TO INSTRUCTION 2, SO DIFFERENT PROCESSES CAN EXECUTE THE SAME PROGRAM AT DIFFERENT RATES.

THIS DRAWING WILL REAPPEAR TWO SLIDES LATER TO ILLUSTRATE THE NOTION OF EACH PROCESS *HAVING ITS OWN DATA AREA.*

PROGRAMS VERSUS PROCESSES

- A PROGRAM IS A SET OF INSTRUCTIONS.
- A PROCESS IS A SET OF ACTIONS CARRIED OUT IN ACCORDANCE WITH INSTRUCTIONS.
- TWO PROCESSES CAN FOLLOW DIFFERENT SETS OF INSTRUCTIONS, OR THEY CAN FOLLOW THE SAME SET OF INSTRUCTIONS, EACH AT ITS OWN PACE.



INSTRUCTOR NOTES

THE TERM <u>RE-ENTRANT</u> ARISES FROM THE FACT THAT AFTER THE PROGRAM HAS BEEN PARTIALLY OR COMPLETELY EXECUTED, IT MAY BE RE-ENTERED TO PRODUCE THE SAME EFFECT.

BECAUSE ALL CODE GENERATED BY AN ADA COMPILER IS GUARANTEED TO BE RE-ENTRANT, ANY ADA SUBPROGRAM (FOR EXAMPLE) CAN BE CALLED BY ONE PROCESS WHILE IT IS STILL BEING EXECUTED BY ANOTHER PROCESS.

SELF-MODIFYING PROGRAMS

- IF A PROGRAM MODIFIES ITSELF, SIMULTANEOUS EXECUTION OF THE PROGRAM BY ASYNCHRONOUS PROCESSES COULD HAVE UNPREDICTABLE EFFECTS.

  -- THE FIRST PROCESS TO ARRIVE AT A CERTAIN POINT CAN CHANGE THE INSTRUCTIONS THAT WILL BE EXECUTED BY THE NEXT PROCESS.

  -- IT IS IMPOSSIBLE TO DETERMINE WHICH PROCESS WILL GET THERE FIRST.

- PROGRAMS AND SUBPROGRAMS TO BE EXECUTED BY MULTIPLE PROCESSES SHOULD NOT MODIFY THEMSELVES.

- PROGRAMS THAT CAN BE SIMULTANEOUSLY EXECUTED BY MULTIPLE PROCESSES ARE CALLED <u>REENTRANT</u>.

- ALL SUBPROGRAMS COMPILED BY AN ADA COMPILER ARE GUARANTEED TO BE REENTRANT AND THEREFORE CANNOT MODIFY THEMSELVES.

INSTRUCTOR NOTES

EACH PERSON IN THE DRAWING IS PERFORMING THE NECESSARY CALCULATIONS ON HIS OWN SCRATCHPAD. THAT IS WHAT ENABLES THEM TO FOLLOW THE SAME INSTRUCTIONS AT THE SAME TIME WITHOUT PAYING ANY ATTENTION TO EACH OTHER.

IF THE INSTRUCTIONS CALLED FOR WRITING FIGURES ON THE SHARED BLACKBOARD AND LATER READING THOSE FIGURES, THE THREE PEOPLE WOULD INTERFERE WITH EACH OTHER'S CALCULATIONS, UNLESS THEY CAREFULLY COORDINATED WITH EACH OTHER.

A NOTE ON IMPLEMENTATION FOR THOSE WHO DO NOT UNDERSTAND HOW PROCESSES EXECUTING THE SAME INSTRUCTIONS CAN HAVE THEIR OWN DATA AREAS:

> TYPICALLY, DATA CAN BE STORED EITHER IN MEMORY -- WHICH IS CONSIDERED TO BE A SINGLE RESOURCE SHARED BY ALL PROCESSORS -- OR IN REGISTERS -- WHICH ARE CONSIDERED TO BE INTEGRAL PARTS OF A PROCESSOR.
>
> EACH VIRTUAL PROCESSOR HAS ITS OWN SET OF VIRTUAL REGISTERS. BEFORE THE RUNTIME SYSTEM SWITCHES THE PROCESSOR TO ANOTHER PROCESS, IT SAVES THE CONTENTS OF THE HARDWARE REGISTERS. BEFORE THE RUNTIME SYSTEM RESUMES EXECUTION OF THAT PROCESS, IT COPIES THE SAVED VALUES BACK INTO THE HARDWARE REGISTERS.
>
> GIVEN THAT THE RUNTIME SYSTEM CREATES THE ILLUSION OF VIRTUAL PROCESSORS EACH WITH ITS OWN SET OF REGISTERS, THE RUNTIME SYSTEM CAN INITIALLY SET ONE OF THOSE REGISTERS TO POINT TO THE DATA AREA IT HAS ALLOCATED FOR THE VIRTUAL PROCESSOR.
>
> THEN TWO PROGRAMS, EACH EXECUTING AN INSTRUCTION TO UPDATE (SAY) THE THIRD WORD AFTER THE ADDRESS IN REGISTER 1 COULD EACH BE UPDATING THE THIRD WORD OF ITS OWN DATA AREA.

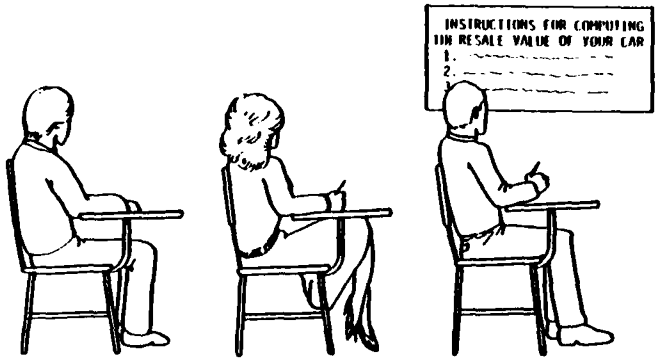
EXCLUSIVE COPIES OF DATA

- WHEN MULTIPLE PROCESSES EXECUTE THE SAME INSTRUCTIONS, THE INTENT IS USUALLY THAT THE EXECUTION OF THE INSTRUCTIONS BY ONE PROCESS NOT AFFECT EXECUTION BY OTHER PROCESSES.

- THIS REQUIRES THAT EACH PROCESS HAVE AN EXCLUSIVE COPY OF THE VARIABLES MODIFIED BY THE INSTRUCTIONS.

  -- THE RUNTIME SYSTEM SUPPLIES A SEPARATE DATA AREA WITH EACH VIRTUAL PROCESSOR.

  -- WHEN A VIRTUAL PROCESSOR EXECUTES INSTRUCTIONS TO EXAMINE OR CHANGE A VARIABLE, IT USES THE COPY OF THE VARIABLE IN ITS OWN DATA AREA.



INSTRUCTOR NOTES

THIS SLIDE SUMMARIZES SECTION 1.

SUMMARY

- AN ADA PROGRAM MAY SPECIFY ACTIONS TO BE PERFORMED BY MORE THAN ONE PROCESS.

  -- EACH PROCESS EXECUTES ACTIONS IN SEQUENCE.
  -- SEVERAL SEQUENCES MAY BE IN PROGRESS AT ONCE.

- THE RUNTIME SYSTEM PROVIDES A VIRTUAL PROCESSOR FOR EACH PROCESS.

  -- VIRTUAL PROCESSORS MAY BE IMPLEMENTED BY INTERLEAVED OR OVERLAPPED CONCURRENCY.

  -- THE RULES OF ADA ALLOW FOR MANY DIFFERENT IMPLEMENTATIONS OF THE RUNTIME SYSTEM, AND DIFFERENT RUNTIME SYSTEMS MAY BE BETTER FOR DIFFERENT APPLICATIONS.

- PROCESSES ARE ASYNCHRONOUS.

  -- THE RELATIVE SPEED OF ONE PROCESS WITH RESPECT TO ANOTHER IS UNPREDICTABLE.

  -- PROCESSES CAN BE SYNCHRONIZED SO THAT THEY WILL BE AT CORRESPONDING SYNCHRONIZATION POINTS AT THE SAME TIME.

- ADA SUBPROGRAMS ARE REENTRANT.

  -- DIFFERENT PROCESSES MAY EXECUTE THE SAME SEQUENCE OF STATEMENTS.

  -- EACH PROCESS HAS AN EXCLUSIVE DATA AREA.

  -- PROGRAMS DO NOT MODIFY THEMSELVES.

INSTRUCTOR NOTES

THIS SECTION PROVIDES A HIGH-LEVEL OVERVIEW OF SITUATIONS IN WHICH CONCURRENCY IS USEFUL.

ALLOW 50-60 MINUTES FOR THIS SECTION.

Section 2

REASONS FOR CONCURRENCY

INSTRUCTOR NOTES

THIS IS AN OUTLINE OF SECTION 2.

EACH OF THE FOUR REASONS FOR CONCURRENCY IS DESCRIBED IN GREATER DETAIL IN COMING SLIDES.

SOME REASONS FOR CONCURRENCY

- MANAGEMENT OF SIMULTANEOUS REAL-WORLD ACTIVITIES
- SIMULATION OF SIMULTANEOUS REAL-WORLD ACTIVITIES
- PARALLEL COMPUTATION TO INCREASE THROUGHPUT
- LOGICAL DECOMPOSITION OF A COMPLEX PROBLEM INTO SIMPLE PROCESSES

INSTRUCTOR NOTES

BULLET 1 DESCRIBES ACTIVITIES GOING ON IN THE REAL WORLD, BULLET 2 DESCRIBES THE CHORES THE PROGRAM MUST PERFORM IN RELATION TO THESE ACTIVITIES. THE CORRESPONDENCE IS AS FOLLOWS:

ACTIVITY 2 ---------------------- CHORE 1
ACTIVITY 3 ---------------------- CHORE 2
ACTIVITIES 4 AND 1 --------------- CHORE 3 (ONE INSTANCE FOR EACH ZONE)
ACTIVITY 5 ---------------------- CHORE 4

ACTIVITIES 4 AND 1 ARE COMBINED BECAUSE MONITORING THE TEMPERATURE AND CONTROLLING THE VENT IN A GIVEN ZONE ARE CLOSELY SYNCHRONIZED ACTIVITIES. EXCEPT FOR THIS, THE REAL-WORLD ACTIVITIES ARE NOT SYNCHRONIZED WITH EACH OTHER. THAT IS ONE REASON THAT IT IS APPROPRIATE TO ASSIGN THE VARIOUS CHORES TO ASYNCHRONOUS PROCESSES.

MANAGEMENT OF SIMULTANEOUS REAL-WORLD ACTIVITIES

- SIMULTANEOUS REAL-WORLD ACTIVITIES FOR A MULTIZONE HEATING SYSTEM:
  - -- TEMPERATURE FLUCTUATES IN EACH ZONE.
  - -- TIME OF DAY CHANGES.
  - -- AT ARBITRARY TIMES, AN OPERATOR KEYS IN DESIRED TEMPERATURE FOR A GIVEN ZONE FOR A GIVEN TIME OF DAY
  - -- VENTS FOR EACH ZONE ARE OPENED AND CLOSED.
  - -- HEATER IS SET TO OFF, LOW, MEDIUM, OR HIGH, DEPENDING ON NUMBER OF OPEN VENTS.

- CONCEPTUALLY, THE PROGRAM MUST SIMULTANEOUSLY PERFORM THE FOLLOWING CHORES:
  - -- KEEP TRACK OF THE TIME OF DAY.
  - -- INTERPRET AND ACT UPON KEYPAD INPUT.
  - -- CONTROL EACH ZONE'S VENT BASED ON CURRENT ZONE TEMPERATURE, TIME OF DAY, AND CURRENT DESIRED TEMPERATURE FOR THE ZONE.
  - -- CONTROL THE HEATER SETTING.

- A SEPARATE PROCESS CAN BE CREATED FOR EACH OF THESE CHORES.
  - -- EACH CHORE HAS ITS OWN SEQUENCE OF INSTRUCTIONS.
  - -- THE PROCESSES MIRROR THE REAL-WORLD ACTIVITIES.

INSTRUCTOR NOTES

TRADITIONALLY, PROBLEMS LIKE THE MULTIZONE HEATING SYSTEM HAVE BEEN HANDLED WITH A CYCLIC EXECUTIVE. THE CYCLIC EXECUTIVE MAKES SURE THAT EACH CHORE IS ATTENDED TO REGULARLY -- ONCE EVERY 500 MSEC IN THIS EXAMPLE. THE EXECUTIVE INITIATES PROCESSING FOR EACH CHORE IN TURN, IN A FIXED ORDER, ALLOCATING A PREDETERMINED PORTION OF THE 500 MSEC TO EACH CHORE. PROCESSING FOR A CHORE IS TERMINATED PREMATURELY IF IT EXCEEDS ITS ALLOTTED TIME LIMIT. EACH 500 MSEC CYCLE IS CALLED A DUTY CYCLE.

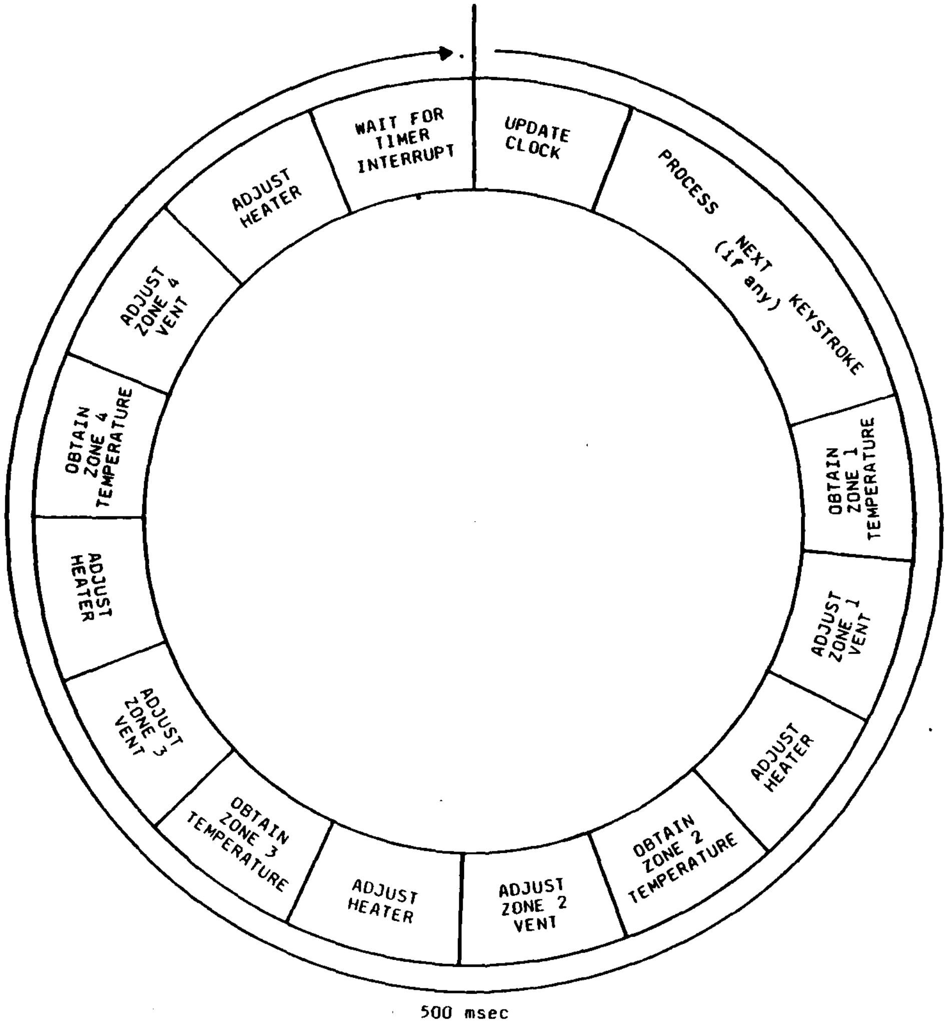
(A VARIATION ON THIS APPROACH IS TO HAVE CERTAIN CHORES PERFORMED IN BACKGROUND. THE PROCESSING FOR THAT CHORE IS ACTIVATED WHENEVER ONE REGULARLY SCHEDULED PIECE OF PROCESSING FINISHES EARLY, AND SUSPENDED WHEN IT IS TIME TO INITIATE THE NEXT SCHEDULED PROCESSING.)

(TRADITIONALLY, THESE "CHORES" ARE CALLED "TASKS." WE HAVE AVOIDED THAT TERM FOR OBVIOUS REASONS.)

SECTION 14 EXAMINES CYCLIC EXECUTIVES MORE CLOSELY.

CYCLIC EXECUTIVES ARE ALSO DISCUSSED IN Ada REAL TIME STUDIES 1986 CHAPTER 2.

A TRADITIONAL VIEW: THE CYCLIC EXECUTIVE



INSTRUCTOR NOTES

WITH ADA, EACH CHORE CAN BE ASSIGNED TO AN ASYNCHRONOUS PROCESS. THEN INTERLEAVING OF PROCESSING FOR EACH CHORE IS MANAGED BY THE RUNTIME SYSTEM.

EACH PROCESS IS RELATIVELY SIMPLE BECAUSE IT IS CONCERNED WITH A SINGLE "THREAD" OF REAL-WORLD ACTIVITY.

IN THIS CASE, EACH REAL-WORLD ACTIVITY IS PERFORMED REPETITIVELY, SO EACH PROCESS EXECUTES AN INFINITE LOOP.

ASSUME THAT THE MULTIZONE HEATING SYSTEM MAINTAINS A SCHEDULE OF DESIRED TEMPERATURES FOR EACH ROOM FOR EACH TIME OF DAY.

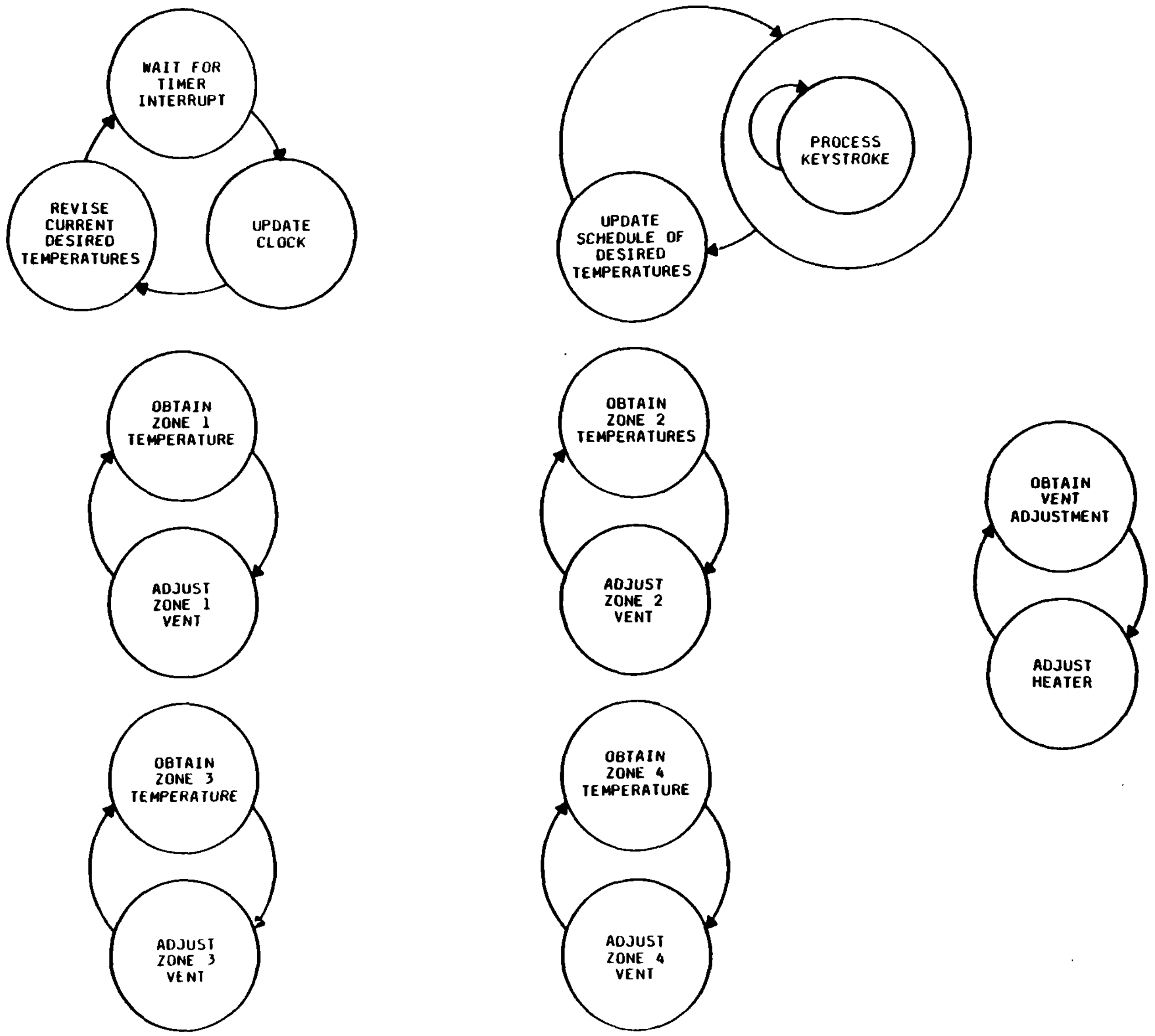
THE FIRST PROCESS KEEPS TRACK OF THE TIME OF DAY. WHENEVER THE TIME IS UPDATED, THIS PROCESS CHECKS WHETHER A TEMPERATURE CHANGE IS NOW SCHEDULED FOR ANY ZONE. IF SO, THAT ZONE'S CURRENT DESIRED TEMPERATURE IS UPDATED.

THE SECOND PROCESS READS AND INTERPRETS KEYSTROKES. WHENEVER A COMPLETE REQUEST IS RECEIVED, THE PROCESS UPDATES THE SCHEDULE OF DESIRED TEMPERATURES ACCORDINGLY. THIS PROCESS CAN BE WRITTEN MUCH LIKE A COMMERCIAL PROGRAM, ANALYZING A STREAM OF INPUT CHARACTERS WITHOUT REGARD TO WHICH CHARACTERS ARE RECEIVED DURING WHICH DUTY CYCLES.

THE PROCESSES FOR THE FOUR ZONES EACH FOLLOW THE SAME INSTRUCTIONS. THEY REPEATEDLY MONITOR THE TEMPERATURE FOR THE ZONE, COMPARE WITH THE ZONE'S CURRENT DESIRED TEMPERATURE, AND OPEN OR CLOSE THE VENT IF NECESSARY. EACH TIME A VENT IS OPENED OR CLOSED, THIS FACT IS REPORTED TO THE LAST PROCESS.

THE LAST PROCESS REPEATEDLY WAITS FOR REPORTS OF VENTS BEING OPENED OR CLOSED, ADJUSTS ITS COUNT OF THE NUMBER OF OPEN VENTS, AND ADJUSTS THE HEATER SETTING.

A SIMPLER VIEW: SINGLE-THREAD PROCESSES


WAIT FOR TIMER INTERRUPT
UPDATE CLOCK
REVISE CURRENT DESIRED TEMPERATURES
PROCESS KEYSTROKE
UPDATE SCHEDULE OF DESIRED TEMPERATURES
OBTAIN ZONE 1 TEMPERATURE
ADJUST ZONE 1 VENT
OBTAIN ZONE 2 TEMPERATURES
ADJUST ZONE 2 VENT
OBTAIN VENT ADJUSTMENT
ADJUST HEATER
OBTAIN ZONE 3 TEMPERATURE
ADJUST ZONE 3 VENT
OBTAIN ZONE 4 TEMPERATURE
ADJUST ZONE 4 VENT

INSTRUCTOR NOTES

BULLET 1: FOR EXAMPLE, ENTERING KEYSTROKES ONE AFTER THE OTHER IS A SINGLE THREAD OF ACTIVITY IN THE REAL WORLD. THE CORRESPONDING PROCESS LOOKS FOR AND PROCESSES A STREAM OF KEYSTROKES AND DOES NOTHING ELSE. THIS IS SIMPLER THAN REPEATEDLY PROCESSING THE KEYSTROKES THAT HAVE ARRIVED IN THE MOST RECENT DUTY CYCLE.

BULLET 2: THE REAL WORLD HAS ACTIVITIES CYCLING SIMULTANEOUSLY. THE CYCLIC EXECUTIVE PROVIDES A DIFFERENT VIEW AND HIDES THE CYCLIC BEHAVIOR OF EACH INDIVIDUAL ACTIVITY.

BULLET 3: INTERACTIONS IN A CYCLIC EXECUTIVE ARE IMPLICIT. IT IS NOT OBVIOUS WHICH SLOTS IN THE DUTY CYCLE CORRESPOND TO THE SAME THREAD OF ACTIVITY. THEREFORE IT IS NOT OBVIOUS WHEN ONE THREAD IS INTERACTING WITH ANOTHER.

ADVANTAGE OF SINGLE-THREAD PROCESSES

- EACH PROCESS IS SIMPLER BECAUSE IT MANAGES A SINGLE THREAD OF ACTIVITY IN THE REAL WORLD.

- SYSTEM STRUCTURE REFLECTS HIGH-LEVEL VIEW OF THE WORLD, NOT LOW-LEVEL CONCERN WITH SCHEDULING.

- INTERACTIONS AMONG THREADS ARE EXPLICITLY REFLECTED IN COMMUNICATION AMONG PROCESSES.

- SIMILAR ACTIVITIES (E.G., MONITORING TEMPERATURE AND CONTROLLING THE VENT IN EACH ZONE) CORRESPOND TO PROCESSES FOLLOWING THE SAME INSTRUCTIONS.

INSTRUCTOR NOTES

THIS IS THE SECOND OF THE FOUR TOPICS LISTED ON SLIDE 2-1.

- BULLET 1: ITEMS 3 AND 4 ARE REAL-TIME SIMULATIONS. ITEMS 1 AND 2 ARE NOT.

  -- ITEM 1: THE SIMULATION MAY INCLUDE RANDOM NUMBER GENERATION WITH A SPECIFIED PROBABILITY DISTRIBUTION.

  -- ITEM 2: DIFFERENT TIMING AND SYNCHRONIZATION OF TRAFFIC LIGHTS CAN BE SIMULATED IN SEARCH FOR A SYSTEM THAT WILL MANIFEST THE DESIRED BEHAVIOR. IT IS LESS EXPENSIVE TO MODIFY THE PROTOTYPE BASED ON SIMULATION THAN TO MODIFY A FULLY IMPLEMENTED SYSTEM BASED ON EXPERIENCE.

  -- ITEM 3: THE FOLLOWING SLIDE EXPANDS UPON THE RADAR EXAMPLE. IN THE SOFTWARE REQUIREMENTS ENGINEERING METHODOLOGY (SREM), A SIMULATION IS RUN BASED ON SOFTWARE REQUIREMENTS, TO TEST WHETHER PERFORMANCE REQUIREMENTS ARE FEASIBLE. (IN THE CASE OF THE RADAR SYSTEM, THE INPUTS TO THE PROGRAM ARE BEING SIMULATED. IN THE CASE OF SREM, EXECUTION OF THE PROPOSED PROGRAM ITSELF IS SIMULATED, BASED ON ASSUMPTIONS ABOUT HOW QUICKLY CERTAIN SPECIFIC FUNCTIONS CAN BE PERFORMED.)

  -- ITEM 4: SIMULATORS CAN BE USED TO TRAIN PEOPLE TO USE NEW EQUIPMENT. THE SIMULATOR SIMULATES THE BEHAVIOR OF THE EQUIPMENT.

- BULLET 2: THE RULES FOR A GIVEN ENTITY MAY DEPEND ON THE CURRENT STATE OF OTHER ENTITIES.

- BULLET 3: THERE IS A ONE-TO-ONE CORRESPONDENCE BETWEEN ENTITIES BEING MODELED AND PROCESSES.

SIMULATING SIMULTANEOUS REAL-WORLD ACTIVITIES

- COMMON USES OF SIMULATION:

  -- PREDICTION OF PHYSICAL PHENOMENA

  - PREDICTION BASED ON AN ASSUMED MODEL (SET OF RULES)
  - MODEL MAY BE TOO COMPLEX FOR MATHEMATICAL ANALYSIS.
  - EXAMPLES: NUCLEAR REACTOR BEHAVIOR, AERODYNAMICS

  -- PROTOTYPING

  - BUILD A PROGRAM SIMULATING A PROPOSED HARDWARE/SOFTWARE SYSTEM
  - OBSERVE THE SYSTEM'S BEHAVIOR AND REFINE THE RULES BEFORE IMPLEMENTING THE SYSTEM
  - EXAMPLE: CENTRAL TRAFFIC SIGNAL CONTROL SYSTEM

  -- SOFTWARE TESTING

  - TEST EMBEDDED COMPUTER *SOFTWARE IN THE LAB BEFORE* INSTALLING IT IN A TANK, AIRCRAFT, OR MISSILE.
  - EXTERNAL INPUTS ARE SIMULATED
  - EXAMPLE: IN A RADAR SYSTEM, SIMULATE AIRCRAFT MOVING IN TRACKS AND GENERATING ECHOES.

  -- TRAINING

  - BUILD A PROGRAM SIMULATING THE SYSTEM BEHIND THE USER INTERFACE.
  - BOTH EXTERNAL EVENTS AND RESPONSES TO USER INPUTS CAN BE SIMULATED.
  - EXAMPLE: COCKPIT SIMULATOR

- IN EACH CASE, SIMULATION IS BASED ON RULES FOR BEHAVIOR OF VARIOUS INDIVIDUAL ENTITIES IN THE MODEL.

- INTERACTIONS ARE COMPLEX, BUT EACH ENTITY CAN BE MODELED BY A PROCESS DIRECTLY IMPLEMENTING THE RULES FOR THAT ENTITY.

INSTRUCTOR NOTES

THE BEHAVIOR SPECIFIED FOR EACH AIRCRAFT WOULD IMPLEMENT SOME SCENARIO.

THE RADAR OUTPUTS WOULD BE EXAMINED TO SEE THAT THEY CORRECTLY DESCRIBED THE SCENARIO.

SOME OF THE AIRCRAFT PROCESSES COULD BE PROGRAMMED TO SIMULATE INTERCEPTION OF ANOTHER AIRCRAFT, FOR EXAMPLE. SOME OF THE "AIRCRAFT" COULD ACTUALLY BE AIR-TO-AIR MISSILES, IN WHICH CASE A FIGHTER PLANE PROCESS WOULD SPAWN A NEW MISSILE PROCESS TO SIMULATE THE LAUNCHING OF A MISSILE.

RADAR SIMULATION EXAMPLE

- TO TEST A RADAR SYSTEM, SIMULATE AIRCRAFT MOVING IN TRACKS AND GENERATING ECHOES.

- PROCESSES TO SIMULATE EACH AIRCRAFT.

  -- EACH PROCESS FOLLOWS INSTRUCTIONS DIRECTLY BASED ON SPECIFICATIONS OF AN INDIVIDUAL AIRCRAFT'S BEHAVIOR.

  -- A PROCESS MAY SIMULATE A SPECIFIED FLIGHT PLAN, JAMMING, DESTRUCTION OF AIRCRAFT, ETC.

- ONE OR MORE PROCESSES TO SIMULATE RADAR HARDWARE.

  -- PROCESS DETERMINES WHAT SIGNALS WOULD BE RECEIVED AT EACH MOMENT, BASED ON POSITION AND VELOCITY OF EACH AIRCRAFT, JAMMING, ETC.

INSTRUCTOR NOTES

THE WHOLE POINT OF SIMULATION IS TO EXPLORE THE CONSEQUENCES OF AN ASSUMED MODEL. THE SIMULATION IS POINTLESS IF IT DOES NOT CORRECTLY IMPLEMENT THE MODEL.

A SIMULATOR PROGRAM BUILT OUT OF PROCESSES IS A DIRECT ENCODING OF A MODEL. IMPLEMENTATION DETAILS DO NOT CLUTTER THE PROGRAM, BUT ARE HIDDEN IN THE RUNTIME SYSTEM.

ADVANTAGES OF CONCURRENT PROCESSES FOR SIMULATION

- EMPHASIS ON EXTERNAL BEHAVIOR, NOT LOW-LEVEL SCHEDULING AND IMPLEMENTATION.

- LIKE THE MODEL ON WHICH THE SIMULATION IS BASED, THE SIMULATION SOFTWARE IS STRUCTURED ACCORDING TO RULES FOR THE BEHAVIOR OF EACH ENTITY.

  -- CORRESPONDENCE BETWEEN MODEL AND SOFTWARE IS EVIDENT AND EASY TO VERIFY.

  -- EASY TO CHANGE SOFTWARE TO MODEL DIFFERENT BEHAVIOR FOR SOME ENTITY.

INSTRUCTOR NOTES

THIS IS THE THIRD OF THE FOUR TOPICS LISTED ON SLIDE 2-1.

BULLET 3: THE TOTAL AMOUNT OF COMPUTATION TIME ON ALL PROCESSORS MAY BE THE SAME, BUT MORE THAN ONE PROCESSOR CAN BE EXPENDING THIS TIME. (THE SAME NUMBER OF PROCESSOR-SECONDS IN FEWER SECONDS.) SLIDES 2-10 AND 2-11 GIVE AN EXAMPLE.

BULLET 4: OTHERWISE, THE PROCESSOR WILL HAVE TO REMAIN IDLE WHILE WAITING FOR THE EXTERNAL EVENT, EVEN THOUGH THERE IS WORK TO BE DONE. SLIDE 2-12 GIVES AN EXAMPLE.

PARALLEL COMPUTATION TO INCREASE THROUGHPUT

- SOMETIMES PARTS OF A COMPUTATION CAN BE DECOMPOSED INTO STEPS THAT DO NOT DEPEND ON EACH OTHERS' RESULTS.

- THESE STEPS CAN BE EXECUTED CONCURRENTLY BY DIFFERENT PROCESSES.

- IF EACH PROCESS IS BEING EXECUTED ON A DIFFERENT PHYSICAL PROCESSOR, THE COMPUTATION CAN COMPLETE MORE QUICKLY.

- IF CERTAIN PROCESSES MUST OCCASIONALLY WAIT FOR EXTERNAL EVENTS TO OCCUR, A SINGLE PROCESSOR CAN COMPLETE THE JOB MORE QUICKLY BY WORKING ON ONE PROCESS WHILE THE OTHER PROCESS IS WAITING.

INSTRUCTOR NOTES

QUICKSORT WAS COVERED IN L305. WE ARE NOT INTERESTED IN DETAILS OF QUICKSORT HERE. WE ARE ESPECIALLY NOT INTERESTED IN HOW STEP 1 IS ACCOMPLISHED, THOUGH STUDENTS SHOULD REALIZE THAT THIS IS WHAT TAKES UP ALMOST ALL OF THE TIME.

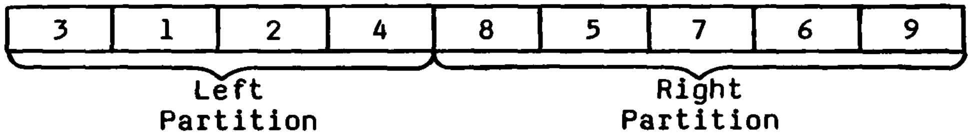
THE NEXT SLIDE DEPICTS THE PARALLELISM.

(FOR SIMPLICITY, WE ARE USING A VERSION OF QUICKSORT DIFFERENT FROM THAT PRESENTED IN L305. IN THE L305 VERSION, THE LEFT AND RIGHT PARTITIONS ARE SEPARATED BY A "SPLITTING ELEMENT" GREATER THAN OR EQUAL TO EACH ELEMENT IN THE LEFT PARTITION AND LESS THAN OR EQUAL TO EACH ELEMENT IN THE RIGHT PARTITION. STEP 1 LEAVES THE SPLITTING ELEMENT IN ITS CORRECT FINAL POSITION.)

EXAMPLE -- PARALLEL SORTING

- QUICKSORT NORMALLY WORKS IN THREE STEPS:

  1. REARRANGE ARRAY ELEMENTS INTO TWO PARTITIONS SUCH THAT EACH ELEMENT IN THE LEFT PARTITION IS LESS THAN OR EQUAL TO EACH ELEMENT IN THE RIGHT PARTITION.



  2. APPLY QUICKSORT RECURSIVELY TO SORT THE LEFT PARTITION.

  3. APPLY QUICKSORT RECURSIVELY TO SORT THE RIGHT PARTITION.

- STEPS 2 AND 3 WORK ON DIFFERENT PARTS OF THE ARRAY, AND CAN PROCEED INDEPENDENTLY IN PARALLEL.

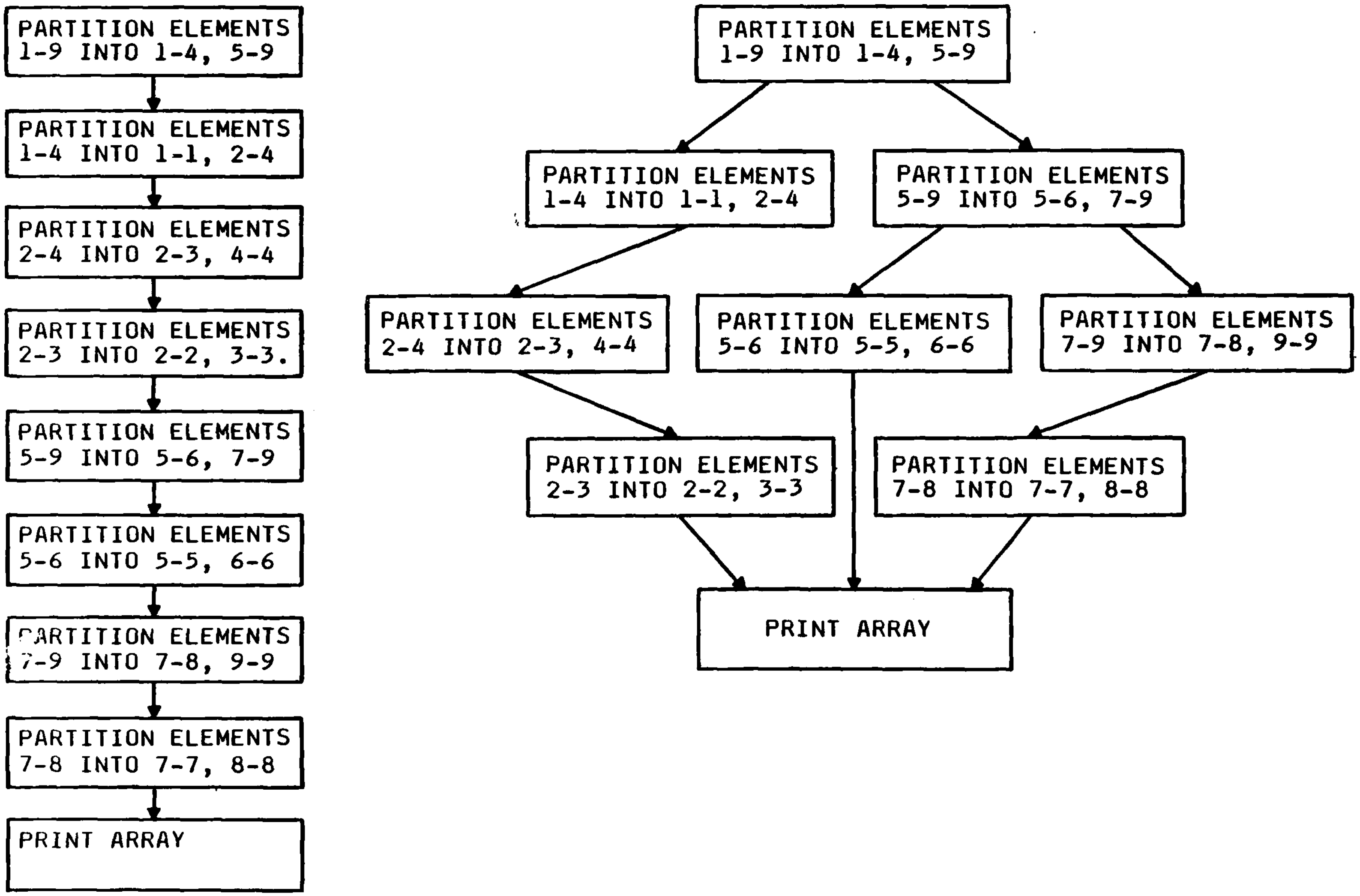
INSTRUCTOR NOTES

THE FIGURE ON THE LEFT SHOWS THE PARTITIONS THAT OCCUR IF STEPS 2 AND 3 PROCEED IN ORDER. THE ORDER OF THE PARTITIONS IS DETERMINED BY THE PATTERN OF RECURSIVE CALLS.

THE FIGURE ON THE RIGHT SHOWS WHAT HAPPENS IF STEPS 2 AND 3 PROCEED IN PARALLEL. AFTER EACH PARTITION, UP TO TWO NEW PARTITIONS BEGIN IN PARALLEL. (PARTITIONS CONSISTING OF ONLY ONE ELEMENT LEAD TO NO FURTHER PARTITIONING.)

BOXES OCCURRING AT THE SAME HEIGHT DO NOT NECESSARILY REPRESENT SIMULTANEOUS EVENTS. FOR EXAMPLE, THE PARTITIONING OF 5-9 MIGHT START WELL AFTER THAT OF 1-4 AND NOT COMPLETE UNTIL THE PARTITIONING OF 2-4 WAS WELL UNDER WAY. HOWEVER, ANY TWO BOXES THAT DO NOT LIE ON THE SAME PATH FROM THE TOP BOX TO THE BOTTOM ONE COULD POSSIBLY BE ACTIVE AT THE SAME TIME.

SEQUENTIAL VERSUS PARALLEL QUICKSORT



PARTITION ELEMENTS 1-9 INTO 1-4, 5-9

PARTITION ELEMENTS 1-4 INTO 1-1, 2-4

PARTITION ELEMENTS 2-4 INTO 2-3, 4-4

PARTITION ELEMENTS 2-3 INTO 2-2, 3-3.

PARTITION ELEMENTS 5-9 INTO 5-6, 7-9

PARTITION ELEMENTS 5-6 INTO 5-5, 6-6

PARTITION ELEMENTS 7-9 INTO 7-8, 9-9

PARTITION ELEMENTS 7-8 INTO 7-7, 8-8

PRINT ARRAY

PARTITION ELEMENTS 1-9 INTO 1-4, 5-9

PARTITION ELEMENTS 1-4 INTO 1-1, 2-4

PARTITION ELEMENTS 5-9 INTO 5-6, 7-9

PARTITION ELEMENTS 2-4 INTO 2-3, 4-4

PARTITION ELEMENTS 5-6 INTO 5-5, 6-6

PARTITION ELEMENTS 7-9 INTO 7-8, 9-9

PARTITION ELEMENTS 2-3 INTO 2-2, 3-3

PARTITION ELEMENTS 7-8 INTO 7-7, 8-8

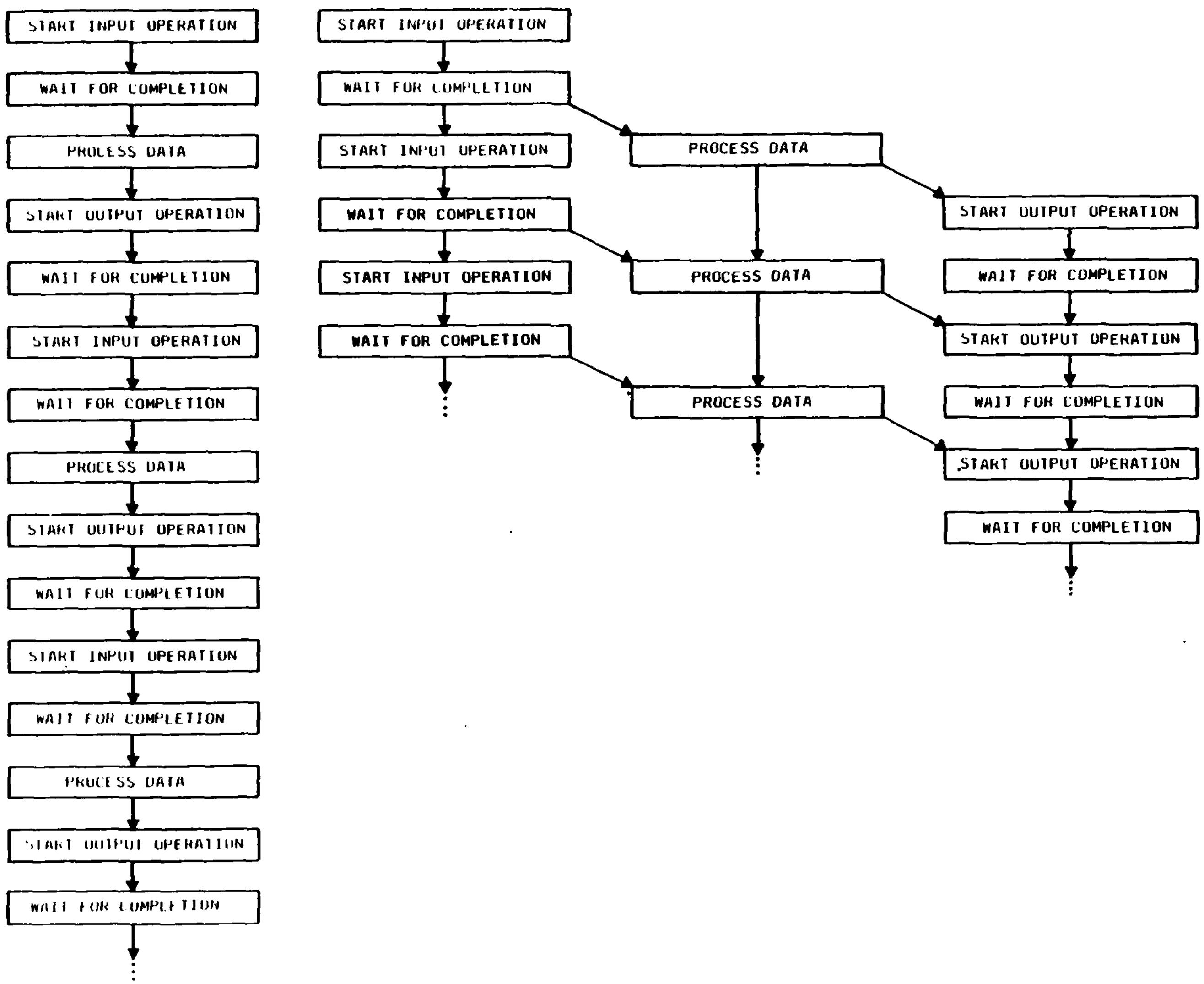
PRINT ARRAY

INSTRUCTOR NOTES

ON THE LEFT, A SINGLE PROCESS EXECUTES A LOOP THAT REPEATEDLY STARTS AN INPUT OPERATION, WAITS FOR THE INTERRUPT SIGNALLING THAT THE OPERATION IS COMPLETE, PROCESSES THE INPUT DATA TO PRODUCE OUTPUT DATA, STARTS AN OUTPUT OPERATION, AND WAITS FOR THE INTERRUPT SIGNALLING THAT THE OUTPUT OPERATION IS COMPLETE.

ON THE RIGHT, ONE PROCESS REPEATEDLY STARTS AN INPUT OPERATION AND WAITS FOR THE INTERRUPT SIGNALLING THAT THE OPERATION IS COMPLETE. IT THEN PASSES THAT DATA TO A SECOND PROCESS, WHICH REPEATEDLY COMPUTES OUTPUT DATA FROM INPUT DATA AND PASSES THE OUTPUT DATA TO A THIRD PROCESS. THE THIRD PROCESS REPEATEDLY ACCEPTS THIS OUTPUT DATA, STARTS AN OUTPUT OPERATION, AND WAITS FOR AN INTERRUPT SIGNALLING THAT THE OUTPUT OPERATION IS COMPLETE. WAITING FOR INPUT, COMPUTING THE OUTPUT FROM A PREVIOUS INPUT, AND WAITING FOR A PREVIOUS OUTPUT TO BE WRITTEN CAN ALL OCCUR SIMULTANEOUSLY.

PARALLEL COMPUTATION TO IMPROVE PROCESSOR UTILIZATION



START INPUT OPERATION
WAIT FOR COMPLETION
PROCESS DATA
START OUTPUT OPERATION
WAIT FOR COMPLETION
START INPUT OPERATION
WAIT FOR COMPLETION
PROCESS DATA
START OUTPUT OPERATION
WAIT FOR COMPLETION
START INPUT OPERATION
WAIT FOR COMPLETION
PROCESS DATA
START OUTPUT OPERATION
WAIT FOR COMPLETION

START INPUT OPERATION
WAIT FOR COMPLETION
START INPUT OPERATION
WAIT FOR COMPLETION
START INPUT OPERATION
WAIT FOR COMPLETION

PROCESS DATA
PROCESS DATA
PROCESS DATA

START OUTPUT OPERATION
WAIT FOR COMPLETION
START OUTPUT OPERATION
WAIT FOR COMPLETION
START OUTPUT OPERATION
WAIT FOR COMPLETION

INSTRUCTOR NOTES

THIS IS THE LAST OF THE FOUR REASONS FOR CONCURRENCY LISTED ON SLIDE 2-1.

THIS USE OF CONCURRENCY IS COVERED IN GREAT DETAIL IN SECTION 15. FOR NOW, PROVIDE A GENERAL OVERVIEW.

THE FOLLOWING SLIDE ILLUSTRATES TRANSFORMATIONS, AND THE SLIDE AFTER THAT RELATES TRANSFORMATIONS TO CONCURRENT PROCESSES.

- BULLET 3:

  THE INPUT/OUTPUT LOGIC OF A TRANSFORMATION CAN BE UNDERSTOOD BY ITSELF. NO CONSIDERATION NEED BE GIVEN TO THE TIMING OF INPUT AND OUTPUT OPERATIONS WITH RESPECT TO OTHER TRANSFORMATIONS.

LOGICAL DECOMPOSITION OF A COMPLEX PROBLEM

- SOMETIMES A COMPLEX COMPUTATION IS MORE EASILY UNDERSTOOD AS A SET OF INTERCONNECTED TRANSFORMATIONS.

- TRANSFORMATIONS ARE CONNECTED BY DATA STREAMS.

  -- EACH TRANSFORMATION OBTAINS DATA FROM AN INPUT STREAM AND DELIVERS DATA TO AN OUTPUT STREAM.

  -- ONE TRANSFORMATION'S OUTPUT STREAM IS ANOTHER TRANSFORMATION'S INPUT STREAM.

  -- DATA PRODUCED BY ONE TRANSFORMATION IS CONSUMED BY ANOTHER.

- TRANSFORMATIONS ARE INDEPENDENT.

  -- EACH CONSUMES INPUT DATA WITHOUT REGARD TO HOW AND WHEN THE DATA WAS PRODUCED.

  -- EACH PRODUCES OUTPUT DATA WITHOUT REGARD TO HOW AND WHEN THE DATA WILL BE CONSUMED.

  -- THIS MAKES EACH TRANSFORMATION EASY TO UNDERSTAND IN ISOLATION.

INSTRUCTOR NOTES

THIS EXAMPLE WAS VERY BRIEFLY PRESENTED IN L305.

(MOTIVATION FOR LINE-LENGTHS: ONE-INCH MARGINS ON 8-1/2 INCH WIDE PAPER ARE PRODUCED BY 65 PICA CHARACTERS OR 78 ELITE CHARACTERS PER LINE. THE DOCUMENT IS BEING REFORMATTED TO BE PRINTED IN ELITE RATHER THAN PICA.)

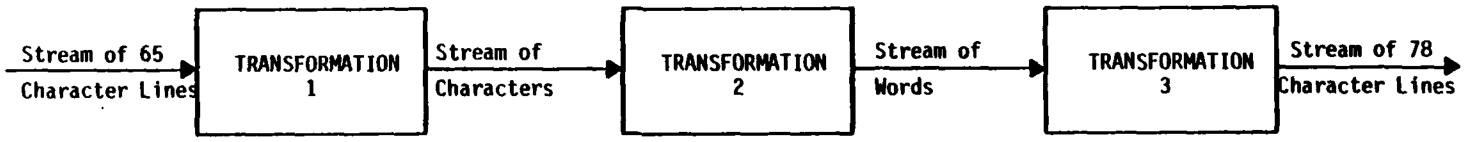
ASSUME THAT PUNCTUATION DIRECTLY ADJACENT TO A WORD, LIKE A PERIOD OR COMMA, IS PART OF A WORD. PUNCTUATION SURROUNDED BY SPACES, SAY A DASH, IS TREATED AS A WORD ITSELF. HYPHENATED WORDS ARE TREATED AS SINGLE WORDS.

TRANSFORMATION 1 ADDS A CHARACTER AFTER EACH LINE SO THAT THE LAST CHARACTER OF THE LINE WILL NOT APPEAR IN THE STREAM OF CHARACTERS TO BE ADJACENT TO THE FIRST CHARACTER OF THE NEXT LINE.

EACH TRANSFORMATION BY ITSELF IS EASILY PROGRAMMED.

EXAMPLE OF DECOMPOSITION INTO TRANSFORMATIONS

- PROBLEM:
  - -- A DOCUMENT HAS BEEN PREPARED WITH LINES THAT ARE 65 CHARACTERS LONG.
  - -- THE DOCUMENT SHOULD BE REFORMATTED WITH LINES THAT ARE 78 CHARACTERS LONG.
  - -- THE NEW DOCUMENT SHOULD FIT AS MANY WORDS ON A LINE AS POSSIBLE.
- DECOMPOSITION INTO THREE TRANSFORMATIONS:
  - -- TRANSFORMATION 1 TAKES A STREAM OF 65-CHARACTER-LINES AS INPUT AND PRODUCES A STREAM OF CHARACTERS AS OUTPUT. THE OUTPUT CONSISTS OF THE CHARACTERS IN EACH LINE, WITH A SPACE ADDED AFTER EACH LINE.
  - -- TRANSFORMATION 2 TAKES THE STREAM OF CHARACTERS PRODUCED BY TRANSFORMATION 1 AS INPUT, GROUPS CONSECUTIVE NON-BLANKS INTO WORDS, AND PRODUCES A STREAM OF WORDS AS OUTPUT.
  - -- TRANSFORMATION 3 TAKES THE STREAM OF WORDS PRODUCED BY TRANSFORMATION 2 AS INPUT, ASSEMBLES WORDS INTO 78-CHARACTER LINES, AND PRODUCES A STREAM OF 78-CHARACTER LINES AS OUTPUT.



INSTRUCTOR NOTES

BULLET 3 REFERS NOT TO THE IMPLEMENTATION OF CONCURRENCY (ABOUT WHICH NOTHING IS BEING ASSUMED), BUT TO THE FACT THAT ONE PROCESS EXPECTING SOME DATUM AS AN INPUT MAY BE UNABLE TO PROCEED UNTIL ANOTHER PROCESS PRODUCES THAT DATUM AS AN OUTPUT. (THESE PROCESSES BEHAVE MUCH LIKE COROUTINES.)

TRANSFORMATIONS AS PROCESSES

- EVEN THOUGH THE ORIGINAL PROBLEM MAY HAVE SEEMED SEQUENTIAL, EACH TRANSFORMATION IS <u>CONCEPTUALLY</u> A CONCURRENT PROCESS.

- IN REALITY, MOST OF THESE PROCESSES ARE USUALLY WAITING FOR ANOTHER PROCESS TO PRODUCE NEEDED DATA.

- VERY LITTLE ACTUAL CONCURRENCY MAY BE LOGICALLY POSSIBLE.

- DECOMPOSITION INTO PROCESSES IS STILL USEFUL.

  -- IT PROVIDES A SIMPLE CONCEPTUAL MODEL OF THE SYSTEM, IN WHICH THE ORDER OF EVENTS CAN BE IGNORED.

  -- EACH PROCESS PERFORMS A SIMPLE SINGLE-THREAD COMPUTATION.

  -- THE RUNTIME SYSTEM HANDLES ALL THE DETAILS OF INTERLEAVING TRANSFORMATIONS, BASED ON WHICH PROCESSES HAVE AVAILABLE THE DATA THEY NEED TO CONTINUE.

INSTRUCTOR NOTES

THIS SLIDE SUMMARIZES THE COMMON THEMES OBSERVED FOR ALL FOUR USES OF CONCURRENCY.

BULLET 4: CONCURRENCY IS A USEFUL CONCEPTUAL MODEL FOR SOLVING A PROBLEM WITH MANY THREADS OF ACTIVITY.

IT IS NOT A LOW-LEVEL IMPLEMENTATION DETAIL. (IN FACT, JACKSON STRUCTURED DESIGN USES CONCURRENCY AS A CONCEPTUAL MODEL EVEN FOR DESIGNING PROGRAMS THAT WILL ULTIMATELY BE IMPLEMENTED WITHOUT CONCURRENCY.)

COMMON THEMES

- SEVERAL PROCESSES MIRROR THE EXISTENCE OF SEVERAL CONCEPTUAL THREADS:

  -- REAL-WORLD ACTIVITIES

  -- ENTITIES BEING SIMULATED

  -- LOGICALLY INDEPENDENT PROCESSING STEPS

  -- TRANSFORMATIONS ACTING ON DATA STREAMS

- DETAILS OF SCHEDULING AND INTERLEAVING ARE HANDLED AUTOMATICALLY BY THE RUNTIME SYSTEM, AND DO NOT APPEAR IN THE PROGRAM.

- THE STRUCTURE OF THE PROGRAM REFLECTS THE CONCEPTUAL THREADS.

- MULTIPROCESS PROGRAMMING IS MORE THAN A WAY OF SPECIFYING THAT CERTAIN ACTIONS CAN BE EXECUTED SIMULTANEOUSLY. IT IS A WAY OF THINKING ABOUT THE STRUCTURE OF A PROBLEM.

INSTRUCTOR NOTES

ALLOW 50 MINUTES LECTURE TIME, THEN PRESENT THE EXERCISE ON SLIDE 3-17.

ALLOW 10 MINUTES BEFORE THE LUNCH BREAK AND 30 MINUTES AFTER IT FOR THE EXERCISE.

Section 3

CONCURRENT PROGRAMMING PROBLEMS

INSTRUCTOR NOTES

- BULLET 1: THE TITLE AND THIS BULLET GIVE THE MAIN MESSAGE OF SECTION 3.

- BULLET 2: EACH OF THESE PROBLEMS IS BRIEFLY DESCRIBED IN THIS SECTION.

  THE DEADLOCK PROBLEM IS EXAMINED MORE CLOSELY IN SECTION 10.

  THERE IS AN ANECDOTE ILLUSTRATING EACH PROBLEM. IF TIME PERMITS, ENCOURAGE STUDENTS TO DESCRIBE THEIR OWN SIMILAR EXPERIENCES.

PROCEED WITH CAUTION!

- CONCURRENT PROGRAMMING IS TRICKY.
- IT ENTAILS MANY SUBTLE PROBLEMS THAT DO NOT ARISE IN SINGLE-PROCESS PROGRAMMING, INCLUDING:
  - -- SIMULTANEOUS UPDATE OF DATA BY MORE THAN ONE PROCESS
  - -- DEADLOCK
  - -- STARVATION
  - -- SYNCHRONIZATION AND COMMUNICATION AMONG PROCESSES
- INTUITION DEVELOPED THROUGH YEARS OF SINGLE-PROCESS PROGRAMMING IS NOT SUFFICIENT.

INSTRUCTOR NOTES

SOUTHERN PATHETIC RAILROAD HAS TWO PARALLEL TRACKS, ONE FOR TRAINS GOING IN EACH DIRECTION.

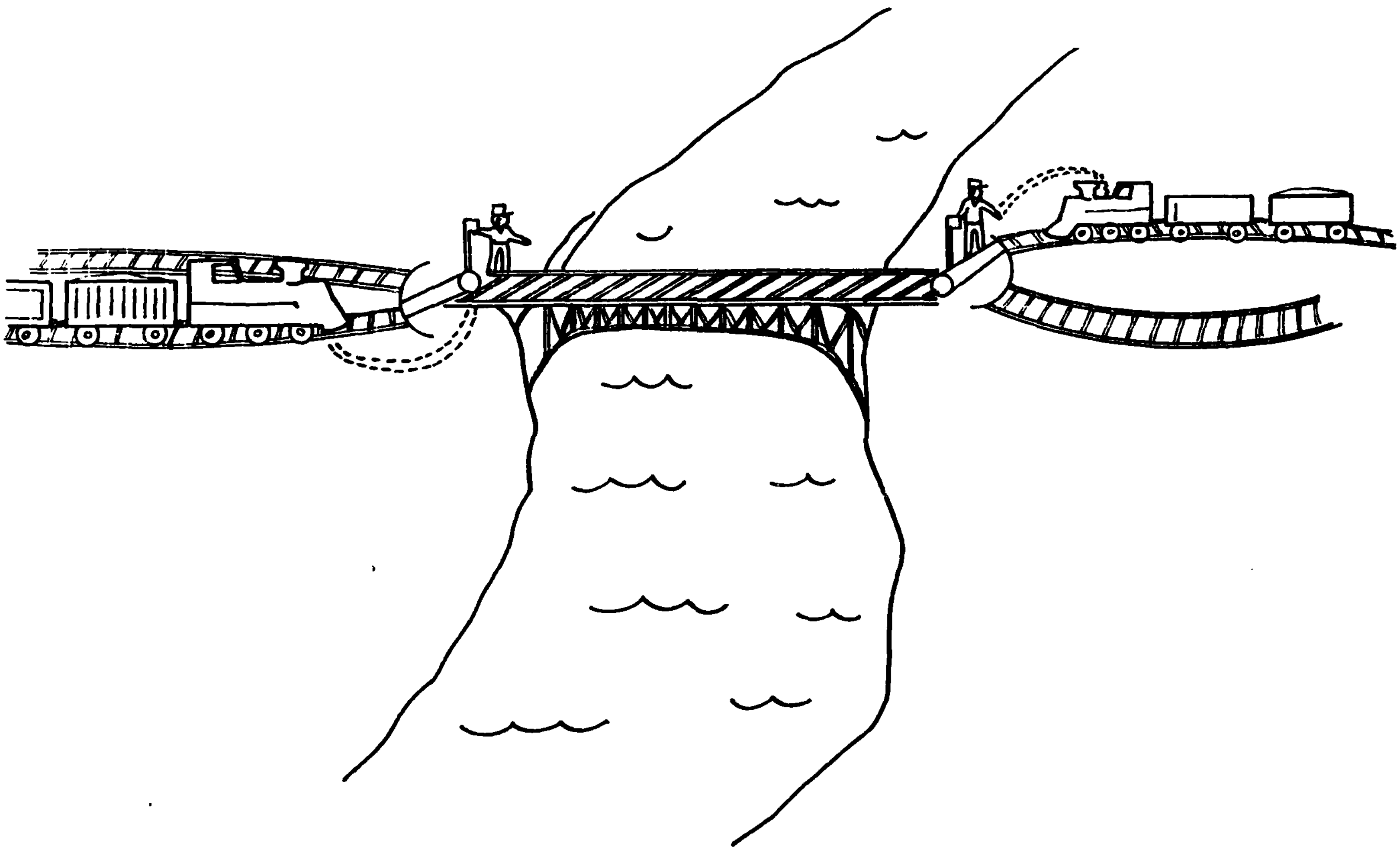
HOWEVER, THERE IS ONLY ONE TRACK CROSSING THE RIVER, SHARED BY TRAINS GOING IN BOTH DIRECTIONS.

SWITCHES AT THE END OF THE SHARED TRACK CONNECT IT WITH EITHER OF THE TWO PARALLEL TRACKS.

IN THIS SCENE, EACH ENGINEER HAS COME TO THE SHARED RAIL, OBSERVED THAT IT WAS NOT (YET) IN USE BY ONCOMING TRAINS, AND SWITCHED THE SHARED TRACK TO HIS DIRECTION. EACH ENGINEER WILL NOW CLIMB BACK IN THE TRAIN AND RESUME HIS TRIP.

WHEN THEY MEET IN THE MIDDLE OF THE BRIDGE, THE ENGINEERS WILL LEARN ABOUT THE HAZARDS OF SIMULTANEOUS UPDATE THE HARD WAY.

SIMULTANEOUS UPDATE



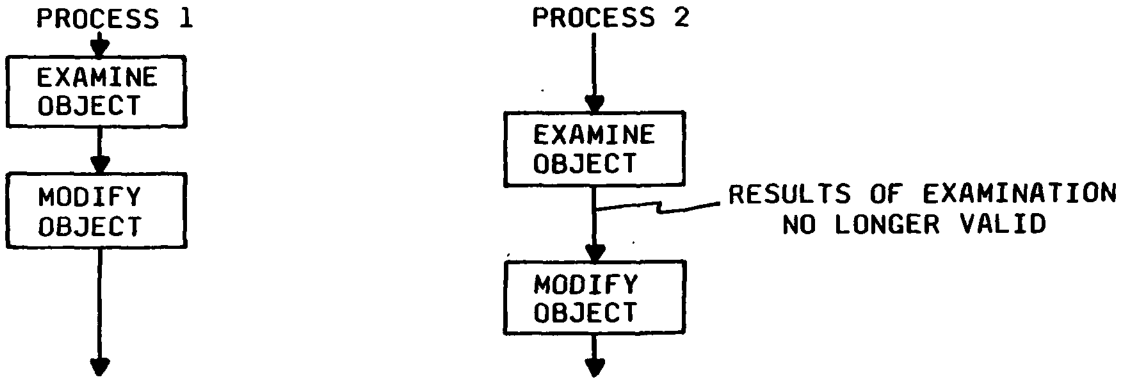
INSTRUCTOR NOTES

ADA HAS FEATURES THAT CAN BE USED TO ACHIEVE MUTUAL EXCLUSION AND THUS AVOID RACE CONDITIONS.

THIS IS ADDRESSED IN SECTION 12.

SIMULTANEOUS UPDATE

- TWO OR MORE PROCESSES EXAMINING AND THEN MODIFYING THE SAME OBJECT.
- RESULTS OF EXAMINATION ARE UNRELIABLE, BECAUSE ANOTHER PROCESS CAN MODIFY THE OBJECT IMMEDIATELY AFTERWARD.



- THIS IS AN EXAMPLE OF A RACE CONDITION. THE OUTCOME OF THE COMPUTATION DEPENDS ON PROCESS TIMING.
- SOLUTION IS MUTUAL EXCLUSION.

-- A PROCESS GAINS EXCLUSIVE ACCESS TO AN OBJECT BEFORE EXAMINING IT AND RELINQUISHES EXCLUSIVE ACCESS AFTER MODIFYING IT.

-- A PROCESS TRYING TO GAIN EXCLUSIVE ACCESS WHILE ANOTHER PROCESS ALREADY HAS IT IS FORCED TO WAIT UNTIL THE OTHER PROCESS RELINQUISHES IT.

INSTRUCTOR NOTES

THIS IS AN ATTEMPT TO PREVENT SIMULTANEOUS UPDATE OF BUFFERS BY ASSIGNING BUFFERS EXCLUSIVELY TO PROCESSES.

IT IS DESIGNED TO WORK AS FOLLOWS:

Buffer_Pool IS A COLLECTION OF 500 BUFFERS, EACH HOLDING 100 FLOAT VALUES. IN ADDITION, EACH BUFFER HAS A FLAG INDICATING WHETHER IT IS CURRENTLY IN USE BY SOME PROCESS.

THE WHILE LOOP SCANS THE BUFFERS LOOKING FOR ONE WHOSE In_Use FLAG IS NOT SET. IF NO SUCH BUFFER IS FOUND, THE SCAN IS CONTINUALLY REPEATED UNTIL SOME OTHER PROCESS RELEASES A BUFFER. (THIS IS CALLED BUSY WAITING BECAUSE THE PROCESS CONTINUES TO CONSUME CPU TIME WHILE IT IS WAITING FOR SOME EXTERNAL EVENT THAT WILL ALLOW IT TO PROCEED. ALTERNATIVE SCHEMES, PRESENTED LATER IN THE COURSE, ALLOW THE PROCESS TO INFORM THE RUNTIME SYSTEM THAT IT IS WAITING FOR SOME EVENT. THE RUNTIME SYSTEM THEN REFRAINS FROM ALLOCATING THE PROCESSOR TO THAT PROCESS UNTIL IT RECEIVES NOTICE FROM SOME OTHER PROCESS THAT THE EVENT HAS OCCURRED. CPU TIME IS NOT WASTED ON HAVING WAITING PROCESSES "SPIN THEIR WHEELS."

ONCE AN APPARENTLY FREE BUFFER IS FOUND, THE PROCESS SETS ITS In_Use FLAG TO MARK IT AS IN USE. THE PROCESS RELEASES THE BUFFER BY RESETTING THE FLAG, SO OTHER PROCESSES MAY USE THE BUFFER.

THIS SCHEME FAILS BECAUSE OF THE POSSIBILITY OF TWO PROCESSES SIMULTANEOUSLY UPDATING THE In_Use FLAGS. FOR EXAMPLE, TWO PROCESSES MIGHT BOTH FIND THAT Buffer_Pool(43).In_Use IS FALSE, BOTH EXIT THE LOOP, AND BOTH SET Buffer_Pool(43).In_Use TO TRUE, SUPPOSEDLY TO PREVENT ANY OTHER PROCESS FROM OBTAINING BUFFER 43.

AN EXAMPLE OF SIMULTANEOUS UPDATE IN PROGRAMMING

- A POOL OF BUFFERS SHARED BY SEVERAL PROCESSES:

```
type Buffered_Data_Type is array (1 .. 100) of Float;
type Buffer_Type is
   record
      In_Use    : Boolean;
      Data_Part : Buffered_Data_Type;
   end record;
Buffer_Pool : array (1 .. 500) of Buffer_Type;
```

- PROCESSES OBTAIN, USE, AND RELEASE BUFFERS AS FOLLOWS:

```
-- Search for an available buffer (busy wait):

I := 1;   -- Each process has its own variable named I
while Buffer_Pool(I).In_Use loop
   if I = Buffer_Pool'Last then
      I := Buffer_Pool'First;  -- Repeat search from start of pool
   else
      I := I + 1;
   end if;
end loop;

-- Use the buffer:
Buffer_Pool(I).In_Use := True;
[operations using Buffer_Pool(I).Data_Part]

-- Release the buffer:
Buffer_Pool(I).In_Use := False;
```

- WHAT IS WRONG WITH THIS SCHEME?

INSTRUCTOR NOTES

SIMULTANEOUS UPDATE -- AN ANECDOTE

- A TEXT EDITOR READS THE CONTENTS OF A FILE INTO A WORKSPACE AT THE BEGINNING OF AN EDITING SESSION AND WRITES THE MODIFIED TEXT BACK TO THE FILE AT THE END OF THE SESSION.

- THE FILE DEVICES.ADA CONTAINS TWO PACKAGES, Sensor_Interface_Package AND Servo_Interface_Package.

- JOHN DOE IS ASSIGNED TO MODIFY Sensor_Interface_Package AND JANE DOE IS ASSIGNED TO MODIFY Servo_Interface_Package.

- THE FOLLOWING EVENTS OCCUR:

  -- JOHN STARTS EDITING DEVICES.ADA, AND THE FILE IS COPIED INTO HIS WORKSPACE.
  -- JANE STARTS EDITING DEVICES.ADA, AND THE FILE IS COPIED INTO HER WORKSPACE.
  -- JOHN BEGINS MODIFYING THE COPY OF Sensor_Interface_Package IN HIS WORKSPACE.
  -- JANE BEGINS MODIFYING THE COPY OF Servo_Interface_Package IN HER WORKSPACE.
  -- JOHN FINISHES HIS WORK AND LEAVES THE EDITOR. THE TEXT IN HIS WORKSPACE IS COPIED BACK TO DEVICES.ADA.
  -- JANE FINISHES HER WORK AND LEAVES THE EDITOR. THE TEXT IN HER WORKSPACE -- INCLUDING THE ORIGINAL VERSION OF Sensor_Interface_Package -- IS COPIED BACK TO DEVICES.ADA.
  -- JOHN AND JANE ARE DIVORCED.

- THIS MARRIAGE COULD HAVE BEEN SAVED IF THE TEXT EDITOR HAD "LOCKED UP" THE FILE AT THE BEGINNING OF AN EDITING SESSION AND UNLOCKED IT AT THE END.

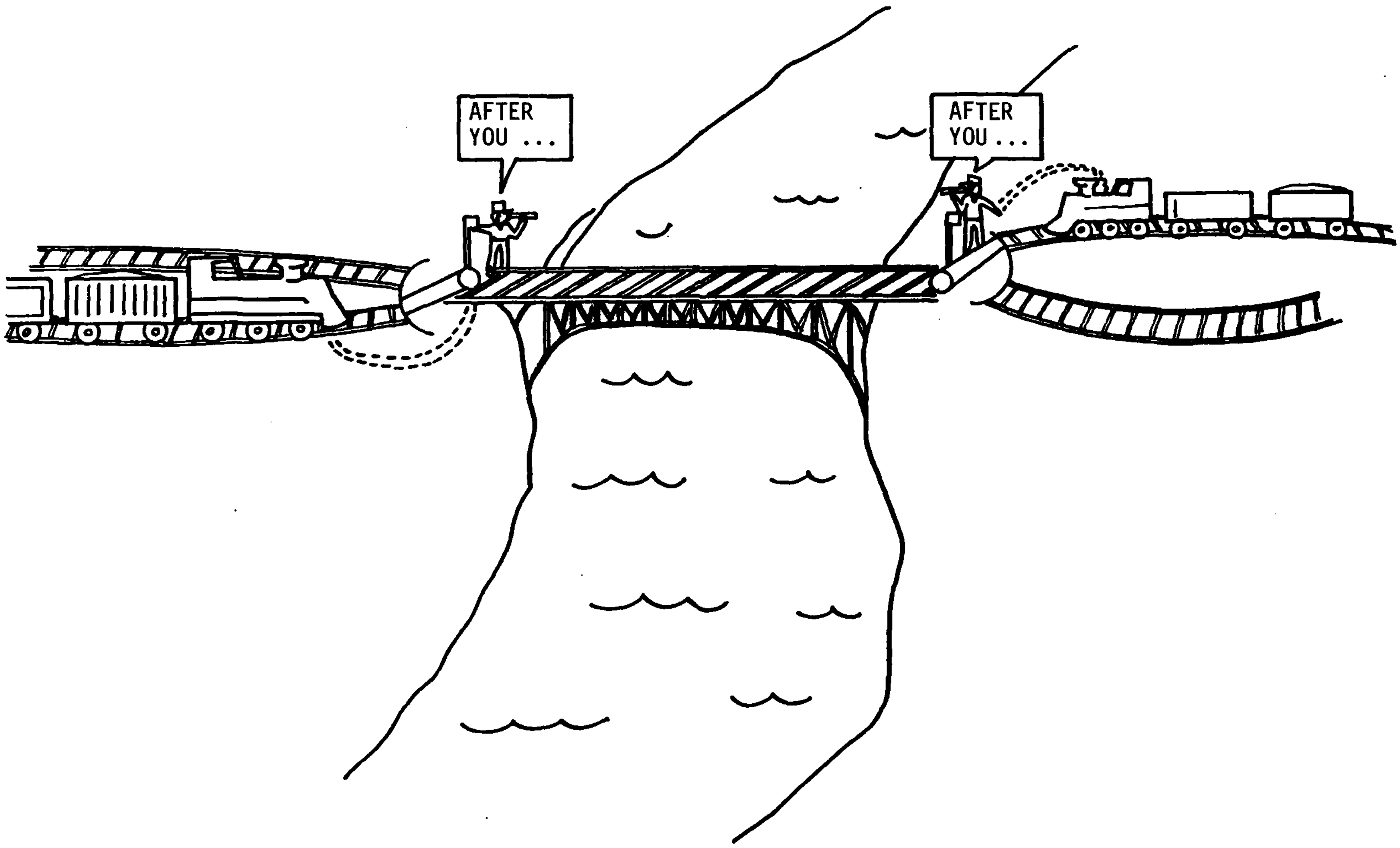
INSTRUCTOR NOTES

TO AVOID A REPETITION OF THE DISASTER DESCRIBED ON SLIDE 3-2, SOUTHERN PATHETIC ISSUED THE FOLLOWING DIRECTIVE TO ITS ENGINEERS:

"UPON COMING TO THE SHARED TRACK, TAKE OUT YOUR TELESCOPE AND CHECK WHETHER THERE IS A TRAIN APPROACHING IN THE OPPOSITE DIRECTION. IF THERE IS, DO NOT PROCEED UNTIL THAT TRAIN HAS PASSED."

THIS SLIDE DEPICTS BOTH ENGINEERS FASTIDIOUSLY ADHERING TO THIS DIRECTIVE, EACH WAITING FOR THE OTHER TO PROCEED.

DEADLOCK



INSTRUCTOR NOTES

- BULLET 1: THE RAILROAD SCENARIO HAD TWO PROCESSES EACH WAITING FOR THE OTHER TO DO SOMETHING.

  IN GENERAL, PATTERNS LIKE THE FOLLOWING ARE POSSIBLE:

  PROCESS 1 WAITS FOR PROCESS 2, WHILE
  PROCESS 2 WAITS FOR PROCESS 3, WHILE
  PROCESS 3 WAITS FOR PROCESS 1.

DEADLOCK

- DEADLOCK OCCURS WHEN THERE IS A CIRCULAR CHAIN OF PROCESSES, EACH OF WHICH IS UNABLE TO PROCEED UNTIL THE NEXT PROCESS IN THE CHAIN DOES SOMETHING.

- THE SYSTEM GRINDS TO A HALT.

- AVOIDING DEADLOCK IS A COMPLEX PROBLEM. SECTION 10 DEALS WITH THIS ISSUE.

INSTRUCTOR NOTES

- BULLET 1: SPOOLING SYSTEMS USUALLY ALSO SPOOL INPUT, BUT THAT DOES NOT CONCERN US HERE.

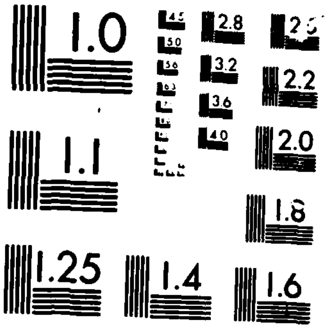
- BULLET 2:
  - ITEM 1: THIS REQUIREMENT IS TO AVOID INTERLEAVING THE OUTPUT OF DIFFERENT JOBS.
  - ITEM 2: THIS REQUIREMENT IS TO MINIMIZE TURNAROUND TIME AND MAXIMIZE PRINTER UTILIZATION.

- BULLET 3: WHAT LOOKS TO THE PROGRAMMER LIKE A REQUEST TO SEND DATA TO THE PRINTER IS HANDLED BY THE OPERATING SYSTEM AS A REQUEST TO THE SPOOLER TO ACCEPT THE DATA.

- BULLET 4: (IT IS NOT NECESSARY TO RESERVE ALL THE NEEDED DISK SPACE IN ADVANCE, BUT ONLY TO ENSURE THAT A JOB WILL *EVENTUALLY* BE ABLE TO OBTAIN ALL THE DISK SPACE IT NEEDS. FOR EXAMPLE, IF JOB A NEEDS 5 TRACKS AND HAS 3, AND JOB B NEEDS 2 TRACKS AND HAS THEM BOTH, JOB A CAN SAFELY BEGIN. SINCE JOB B HAS ALL THE DISK SPACE IT NEEDS, IT WILL EVENTUALLY COMPLETE, MAKING AVAILABLE THE REST OF THE DISK SPACE THAT A NEEDS TO COMPLETE. THE "BANKER'S ALGORITHM," DESCRIBED IN BRINCH-HANSEN'S *OPERATING SYSTEM PRINCIPLES*, IS BASED ON THIS IDEA.)

AD-A166 351 ADA (TRADE NAME) TRAINING CURRICULUM REAL-TIME SYSTEMS 2/5
IN ADA L401 TEACHER'S GUIDE VOLUME 1(U) SOFTECH INC
WALTHAM MA 1986 DAAB07-83-C-K514

UNCLASSIFIED F/G 5/9 NL



MICROCOPY RESOLUTION TEST CHART

AN EXAMPLE OF DEADLOCK IN PROGRAMMING

- OPERATING SYSTEMS TYPICALLY CONTAIN OUTPUT SPOOLERS.
  - USER PROGRAMS WRITE PRINTER OUTPUT TO A DISK.
  - ONCE A PROGRAM IS COMPLETE AND THE PRINTER IS AVAILABLE, THE SPOOLER COPIES THE PROGRAM'S OUTPUT TO THE PRINTER.
- THE SPOOLER DOES NOT BEGIN PRINTING A PROGRAM'S OUTPUT UNTIL THE PROGRAM IS COMPLETE.
  - ONCE THE PRINTER BEGINS PRINTING A PROGRAM'S OUTPUT, IT MUST DO SO IN ITS ENTIRETY BEFORE GOING ON TO ANOTHER PROGRAM'S OUTPUT.
  - THE PRINTER SHOULD NOT BE KEPT IDLE WAITING FOR ITS CURRENT JOB TO PRODUCE ANOTHER OUTPUT LINE WHILE OTHER JOBS COMPLETE AND WAIT TO BE PRINTED.
- DEADLOCK CAN OCCUR WHEN THE SPOOLER RUNS OUT OF DISK SPACE.
  - USER PROGRAMS WAIT FOR THE SPOOLER TO ACCEPT MORE OUTPUT.
  - THE SPOOLER WAITS FOR A USER PROGRAM TO COMPLETE SO THAT ITS OUTPUT CAN BE PRINTED, DISK SPACE CAN BE FREED, AND MORE OUTPUT CAN BE ACCEPTED.
- ONE SOLUTION IS TO FORCE EACH JOB TO SPECIFY THE MAXIMUM NUMBER OF OUTPUT LINES IT CAN PRODUCE.
  - A JOB CANNOT BEGIN EXECUTING UNTIL THE SPOOLER IS ABLE TO ENSURE THAT ENOUGH DISK SPACE WILL BE AVAILABLE FOR THE JOB TO COMPLETE.
  - A PROGRAM IS ABORTED IF IT ATTEMPTS TO EXCEED ITS STATED MAXIMUM.

INSTRUCTOR NOTES

THE EVENTS YOU ARE ABOUT TO RELATE ARE TRUE. ONLY THE NAMES HAVE BEEN CHANGED TO PROTECT THE INNOCENT.

(IT HAPPENED TO ONE OF THE AUTHORS OF THIS MODULE WHILE THE MODULE WAS BEING DESIGNED.)

DEADLOCK -- AN ANECDOTE

MS. SMITH	MR. JONES' SECRETARY
PICKS UP PHONE AND DIALS JONES	
	NOTICES JONES' PHONE IS OFF THE HOOK.
	ANSWERS CALL FROM SMITH.
"HELLO, IS MR. JONES IN?"	
	"YES, BUT HE'S ON ANOTHER LINE. WOULD YOU LIKE TO HOLD?"
"YES, PLEASE."	
	PLACES SMITH ON HOLD.
STAYS ON THE LINE WAITING FOR JONES TO FINISH HIS CALL.	
↓	

MR. JONES	MS. SMITH'S SECRETARY
PICKS UP PHONE AND DIALS SMITH	
	NOTICES SMITH'S PHONE IS OFF THE HOOK.
	ANSWERS CALL FROM JONES
"GOOD MORNING, I'D LIKE TO SPEAK TO MS. SMITH, PLEASE."	
	"I'M AFRAID SHE'S TIED UP ON ANOTHER LINE RIGHT NOW. CAN I TAKE A MESSAGE OR WOULD YOU LIKE TO HOLD?"
"I THINK I'LL HOLD."	
	PLACES JONES ON HOLD.
STAYS ON LINE WAITING FOR SMITH TO FINISH HER CALL.	
↓	

INSTRUCTOR NOTES

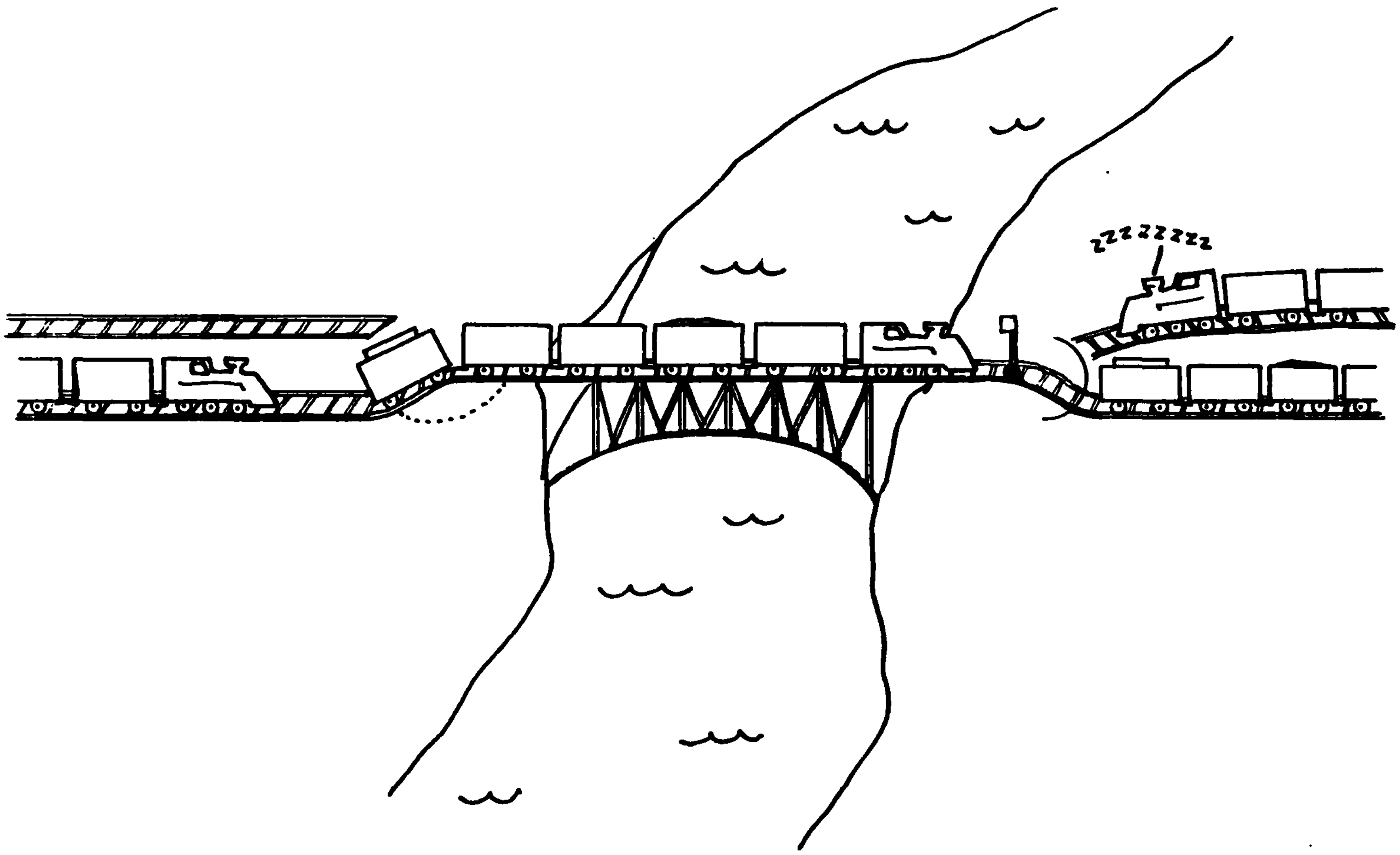
AFTER STUDYING THE DEADLOCK PROBLEM DESCRIBED ON SLIDE 3-6, SOUTHERN PATHETIC RAILROAD REVISED ITS DIRECTIVE AS FOLLOWS:

> "UPON APPROACHING THE SHARED TRACK FROM THE WEST, AN EASTBOUND TRAIN SHALL WAIT FOR ANY WESTBOUND TRAIN <u>ALREADY USING</u> THE SHARED TRACK TO PASS, AND THEN PROCEED."
>
> "UPON APPROACHING THE SHARED TRACK FROM THE EAST, A WESTBOUND ENGINEER SHALL TAKE OUT HIS TELESCOPE AND WAIT FOR ANY EASTBOUND TRAIN <u>APPROACHING</u> THE SHARED TRACK TO PASS BEFORE PROCEEDING."

IN OTHER WORDS, EASTBOUND TRAINS HAVE PRIORITY OVER WESTBOUND TRAINS WHEN THERE IS CONTENTION FOR USE OF THE TRACK.

THIS SLIDE DEPICTS A WESTBOUND TRAIN WHOSE ENGINEER HAS FALLEN ASLEEP WHILE WAITING FOR A TIME AT WHICH NO EASTBOUND TRAIN IS APPROACHING. THE WESTBOUND TRAIN NEVER GETS A TURN TO USE THE SHARED TRACK.

STARVATION



INSTRUCTOR NOTES

THIS MAY BE A GOOD POINT AT WHICH TO REASSURE STUDENTS THAT THERE WILL BE A BREAK FOR LUNCH AFTER JUST A FEW MORE SLIDES.

THE NEXT SLIDE GIVES SOME SPECIFIC SCENARIOS FOR STARVATION.

STARVATION

- STARVATION OCCURS WHEN CERTAIN PROCESSES NEVER GET A CHANCE TO EXECUTE.
- IMPORTANT PROCESSING MAY NEVER TAKE PLACE.
- STARVATION IS DIFFERENT FROM DEADLOCK.

  -- A STARVED PROCESS MAY NOT HAVE VOLUNTARILY BEGUN WAITING FOR AN EVENT.

  -- ALL OTHER PROCESSES MAY APPEAR TO BE WORKING NORMALLY.

  -- STARVATION OFTEN APPEARS TO BE A QUIRK OF ASYNCHRONOUS SCHEDULING, THOUGH THERE MAY BE A SYSTEMATIC CAUSE.
- STARVATION MAY BE CAUSED BY A RUNTIME SYSTEM OR BY PROGRAMMING.

INSTRUCTOR NOTES

- BULLET 1:

  ITEMS 1 AND 2 ARE EXTREME EXAMPLES, BUT FAIRNESS IS AN IMPORTANT CRITERION IN EVALUATING VARIOUS RUNTIME SYSTEMS.

- BULLET 2:

  PRIORITIES ARE DISCUSSED FORMALLY IN SECTION 20. FOR NOW, RELY ON STUDENTS' INTUITIVE UNDERSTANDING OF WHAT PRIORITIES ARE.

  - ITEM 1: THIS IS A SUMMARY OF WHAT PRIORITIES MEAN IN ADA.

  - ITEM 3: THIS IS A WAY OF OVERRIDING A PRIORITY WHEN IT HAS STARVED ANOTHER PROCESS TOO LONG. WHEN A PROCESS IS BLOCKED (FORCED TO WAIT FOR SOME EVENT) ITS PRIORITY IS IRRELEVANT. AN EXAMPLE WILL BE GIVEN IN SECTION 20. (THE PRIORITY OF THE STARVED PROCESS CANNOT SIMPLY BE RAISED, BECAUSE THE PRIORITY ASSIGNED TO AN ADA TASK IS PERMANENT.)

- BULLET 4:

  - ITEM 3: GIVEN A TEMPORARY PROCESSING OVERLOAD THAT DOES NOT ALLOW ALL ACTIVE TRACKS TO BE KEPT CURRENT, THE BEST WAY TO CONTINUE MAY BE TO SACRIFICE CURRENT DATA ABOUT FAR-AWAY TRACKS SO THAT DATA ABOUT NEARBY TRACKS CAN BE KEPT CURRENT.

EXAMPLES OF STARVATION IN PROGRAMMING

- AN "UNFAIR" RUNTIME SYSTEM
  - -- LAST-IN FIRST-OUT SCHEDULING "QUEUE" FOR PROCESSES
  - -- NO PREEMPTION UNLESS A HIGHER-PRIORITY PROCESS BECOMES ELIGIBLE FOR EXECUTION.
  - -- TECHNICALLY, SUCH RUNTIME SYSTEMS CONFORM TO THE RULES OF ADA, BUT THEY ARE NOT USEFUL FOR REAL-TIME PROGRAMMING.
- STARVATION OF LOW PRIORITY PROCESSES
  - -- A LOW-PRIORITY PROCESS IS NOT ALLOWED TO EXECUTE WHILE A HIGH-PRIORITY PROCESS WAITS FOR A PROCESSOR.
  - -- TO AVOID STARVATION, HIGH PRIORITIES SHOULD BE RESERVED FOR PROCESSES PERFORMING SHORT, URGENT CHORES AT (RELATIVELY) INFREQUENT INTERVALS.
  - -- A PROGRAM CAN BE DESIGNED SO THAT A HIGH-PRIORITY PROCESS BECOMES TEMPORARILY BLOCKED AFTER A LOW-PRIORITY PROCESS HAS BEEN WAITING FOR TOO LONG.
- "CLIQUES" OF PROCESSES
  - -- THREE PROCESSES EACH REPEATEDLY WAIT FOR A MESSAGE FROM EITHER OF THE OTHER PROCESSES AND RESPOND WITH A RETURN MESSAGE TO THAT PROCESS.
  - -- THE FIRST TIME ONE PROCESS SENDS A MESSAGE TO ANOTHER, THEY WILL START A "CONVERSATION" THAT PERMANENTLY EXCLUDES THE THIRD PROCESS.
- ONE PROCESS GIVING UNEQUAL SERVICE TO OTHER PROCESSES.
  - -- A RADAR SYSTEM MAY CONSIST OF A CONTROLLING PROCESS PLUS ONE PROCESS FOR EACH ACTIVE TRACK.
  - -- THE CONTROLLING PROCESS MAY UNBLOCK EACH TRACK PROCESS IN TURN, STARTING WITH THE INNERMOST AND WORKING OUTWARD, BUT STARTING WITH THE INNERMOST AGAIN AFTER EACH SWEEP.
  - -- PROCESSES FOR THE OUTERMOST TRACKS MAY BE STARVED WHEN THE SYSTEM IS BUSY, BUT THIS MAY BE DESIRABLE.

INSTRUCTOR NOTES

A SOLUTION (ASSUMING THAT PROCESSES ARE CREATED INFREQUENTLY ENOUGH) IS TO CONSIDER PROCESS IDENTIFICATION NUMBER x GREATER THAN PROCESS IDENTIFICATION NUMBER y WHENEVER EITHER

$$x > y \text{ AND } x - y < 5000$$

$$x < y \text{ AND } y - x \geq 5000$$

THAT IS, THINK OF PROCESS IDENTIFICATION NUMBERS AS IF ARRANGED ON A CLOCK DIAL, WITH 0 FOLLOWING 9999, AND CONSIDER THE SHORTER OF THE TWO ARCS CONNECTING x AND y. WHICHEVER PROCESS IDENTIFICATION NUMBER APPEARS AT THE BEGINNING OF THE SHORTER ARC (GOING CLOCKWISE) IS CONSIDERED SMALLER.

STARVATION -- AN ANECDOTE

- A COMMERCIAL OPERATING SYSTEM ASSIGNS "PROCESS IDENTIFICATION NUMBERS" TO PROCESSES IN THE ORDER THEY ARE CREATED.
- WHEN A VIRTUAL PROCESSOR BECOMES AVAILABLE, IT IS ASSIGNED TO RUN THE PROCESS WITH THE LOWEST IDENTIFICATION NUMBER.
- BY THE TIME PROCESS IDENTIFICATION NUMBER 9999 IS REACHED, PROCESS 0 HAS LONG SINCE COMPLETED, SO ASSIGNMENT OF NUMBERS CYCLES BACK TO 0.
- ONCE NUMBERS LIKE 0, 1, 2, ... BEGIN TO BE ASSIGNED, QUEUED PROCESSES WITH NUMBERS IN THE 9900'S MAY BE STARVED FOR A VERY LONG TIME.

INSTRUCTOR NOTES

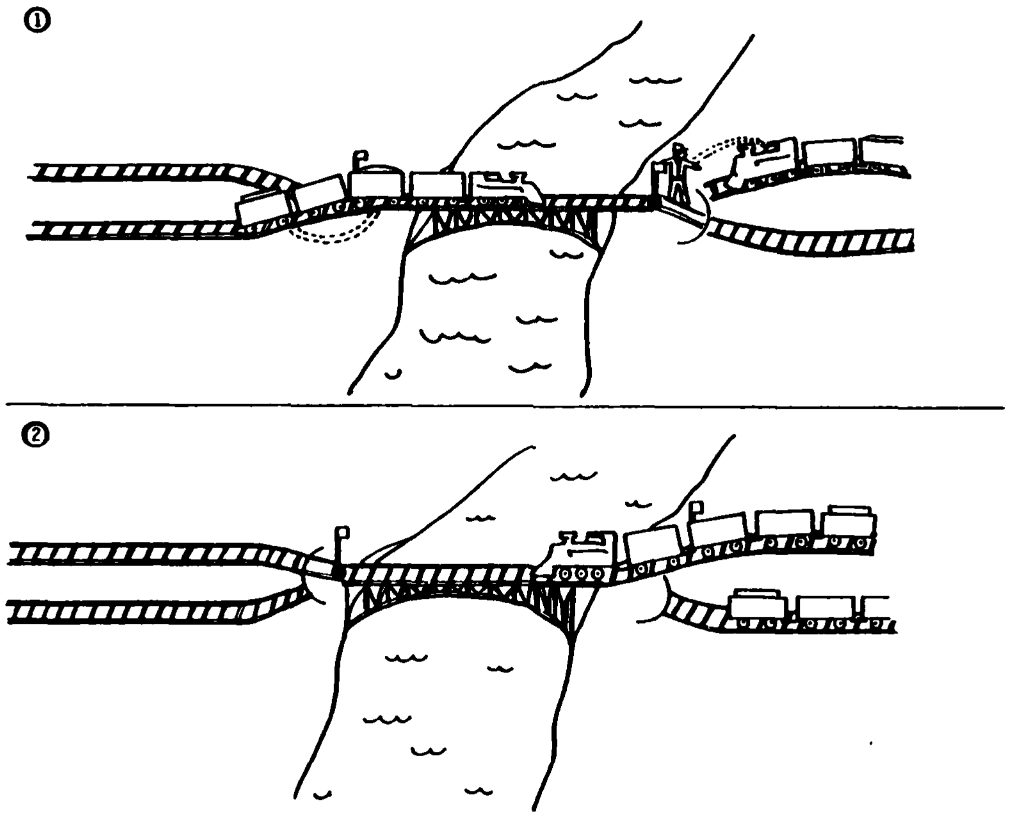
AFTER THE WESTBOUND RIDERS ASSOCIATION TIRED OF HALTING (W.R.A.T.H.) SUED SOUTHERN PATHETIC RAILROAD, THE RAILROAD ADOPTED A NEW SCHEME.

TRAINS IN EACH DIRECTION NOW HAVE AN EQUAL CHANCE OF GETTING TO USE THE SHARED TRACK FIRST. THERE IS STILL NO CHANCE OF COLLISION, AND VIRTUALLY NO CHANCE OF PERMANENT DEADLOCK (THOUGH IT MAY TAKE A WHILE TO DETERMINE WHICH TRAIN GOES FIRST).

THE RAILROAD NOW RUNS SAFELY, EFFICIENTLY, AND FAIRLY.

(IN A FEW MINUTES, STUDENTS WILL BE GIVEN AN EXERCISE TO DETERMINE HOW THIS WAS DONE.)

PROCESS COOPERATION



INSTRUCTOR NOTES

IN ADA, COMMUNICATION AND SYNCHRONIZATION ARE BOTH ADDRESSED BY THE RENDEZVOUS MECHANISM, WHICH IS INTRODUCED IN SECTION 7.

ALL OF PART III (SECTIONS 7-10) IS DEVOTED TO THE PROBLEM OF MAKING PROCESSES WORK COOPERATIVELY.

PROCESS COOPERATION

- SIMULTANEOUS UPDATE, DEADLOCK, AND STARVATION WOULD BE NO PROBLEM IF PROCESSES WORKED IN IGNORANCE OF EACH OTHER. BECAUSE PROCESSES MUST WORK IN COOPERATION, THERE ARE TWO CONCERNS:

- SYNCHRONIZATION

    -- FORCING ACTIONS PERFORMED BY DIFFERENT PROCESSES TO OCCUR IN A SPECIFIED ORDER.

    -- PREVENTING ONE PROCESS FROM GETTING AHEAD OF OR FALLING BEHIND OTHER PROCESSES.

    -- PREVENTING SIMULTANEOUS UPDATE.

    -- CONSTRAINING THE INTERLEAVING OF PROCESSES AS LITTLE AS POSSIBLE

    - TO ALLOW MAXIMAL CPU UTILIZATION AND PARALLELISM
    - TO AVOID DEADLOCK AND STARVATION

- COMMUNICATION

    -- LETTING ONE PROCESS USE DATA PRODUCED BY ANOTHER PROCESS

    -- PREVENTING SIMULTANEOUS UPDATE

    -- LETTING DIFFERENT PROCESSES MODIFY THE SAME DATA (AT DIFFERENT TIMES)

    -- ENSURING THAT DATA CAN BE PRODUCED BY ONE PROCESS AS FAST AS (BUT NOT FASTER THAN) IT CAN BE CONSUMED BY ANOTHER

INSTRUCTOR NOTES

DON'T WORRY IF THE STUDENTS (OR YOU, FOR THAT MATTER) DO NOT UNDERSTAND ALL THE INTRICACIES OF THIS PROBLEM. THE WHOLE POINT IS THAT TASK SYNCHRONIZATION CAN BE COMPLEX, AND THAT TASKS CAN INTERACT IN SURPRISING WAYS THAT ARE HARD TO UNDERSTAND.

THE FIRST THREE BULLETS DESCRIBE THREE SEPARATE ASPECTS OF THE SHUTTLE SOFTWARE. THE FOURTH BULLET DESCRIBES AN UNANTICIPATED CONSEQUENCE OF THEIR INTERACTION.

THIS PROBLEM IS DISCUSSED IN GREAT DETAIL IN JOHN GARMAN'S ARTICLE, "THE BUG HEARD ROUND THE WORLD", WHICH IS THE SOURCE OF THE INFORMATION ON THIS SLIDE.
(SEE BIBLIOGRAPHY.)

PROCESS COOPERATION -- AN ANECDOTE

- IN SOFTWARE FOR THE SPACE SHUTTLE, PROCESSOR OVERLOADS OCCUR ROUTINELY.
  - -- SYMPTOM IS AN ATTEMPT TO SCHEDULE AN EVENT FOR A TIME THAT HAS PASSED.
  - -- RESPONSE IS TO RESCHEDULE THE EVENT FOR THE CORRESPONDING TIME SLOT IN THE NEXT DUTY CYCLE.
- THE SHUTTLE SOFTWARE USES ITS SCHEDULING QUEUE AS A CLOCK.
  - -- HUNDREDS OF EVENTS ARE SCHEDULED EACH SECOND, SO THE TIME OF THE NEXT SCHEDULED EVENT IS A FAIRLY ACCURATE ESTIMATE OF THE CURRENT TIME.
- WHEN THE SYSTEM IS INITIALIZED, CERTAIN CYCLIC PROCESSING MUST BEGIN AT A PARTICULAR TIME SO THAT IT WILL BE IN PHASE WITH THE TELEMETRY SYSTEM.
- RARELY, THIS TIME TURNS OUT TO BE EARLIER THAN THE TIME OF THE FIRST EVENT IN THE SCHEDULING QUEUE.
  - -- SINCE THE TIME OF THE FIRST SCHEDULED EVENT IS TAKEN AS THE CURRENT TIME, THE CYCLIC PROCESSING STARTUP TIME APPEARS TO BE IN THE PAST.
  - -- THE START OF THIS CYCLIC PROCESSING IS RESCHEDULED, AND THE PROCESSING FALLS ONE CYCLE BEHIND THE REST OF THE SYSTEM.
- THIS HARDLY EVER HAPPENS.
  - -- IT DEPENDS ON THE POINT IN THE TELEMETRY SYSTEM CYCLE AT WHICH THE COMPUTER SYSTEM IS TURNED ON.
  - -- THERE IS ONLY A POTENTIAL PROBLEM DURING 1.5% OF THE TELEMETRY SYSTEM CYCLE.
  - -- IT HAPPENED ON APRIL 10, 1981, THE DAY ON WHICH THE FIRST SHUTTLE LAUNCH WAS SCHEDULED.
  - -- THE COUNTDOWN CAME TO A HALT WITH 20 MINUTES TO GO BECAUSE A COMPUTER RUNNING BACKUP SOFTWARE COULD NOT SYNCHRONIZE WITH THE PRIMARY SYSTEM, WHICH WAS NOT IN SYNCH WITH ITSELF.

INSTRUCTOR NOTES

INTRODUCE THE EXERCISE, ALLOWING TEN MINUTES TO EXPLAIN IT, ANSWER QUESTIONS, AND GET STUDENTS STARTED THINKING.

MAKE SURE STUDENTS UNDERSTAND THAT TO WAIT UNTIL A CONDITION C BECOMES TRUE, WAIT SHOULD BE USED IN A LOOP OF THE FORM

```
while not C loop
  Wait;
end loop;
```

THEN LET THE STUDENTS BREAK FOR LUNCH, SUGGESTING THAT THEY CONSIDER THE PROBLEM OVER LUNCH.

ALLOCATE 30 MINUTES AFTER LUNCH TO PROBLEM SOLVING, PRESENTATION OF SOLUTIONS, AND DISCUSSION OF SOLUTIONS, DIVIDING THE TIME ACCORDING TO THE INSTRUCTOR'S DISCRETION. THIS IS A DIFFICULT PROBLEM, AND EVEN TALENTED AND EXPERIENCED STUDENTS MAY BE UNABLE TO SOLVE IT. THE IMPORTANT THING IS TO GET STUDENTS TO THINK ABOUT THE RELEVANT ISSUES AND APPRECIATE THE SUBTLETY OF CONCURRENT PROGRAMMING. FOR THIS REASON, AN INSTRUCTOR WHO FEELS COMFORTABLE DISCUSSING, STUDENT SOLUTIONS "ON THE FLY" MIGHT WISH TO CUT OFF THE PROBLEM SOLVING AND BEGIN DISCUSSION AFTER TEN OR FIFTEEN MINUTES.

THE SOLUTION DESCRIBES BOTH A CORRECT APPROACH AND SEVERAL NEAR-MISSES REFLECTING COMMON ERRORS.

EXERCISE 3.1

- SOUTHERN PATHETIC RAILROAD HAS HIRED YOU TO DEVELOP A SCHEME THAT WILL ALLOW TRAINS TO SHARE A TRACK SAFELY, EFFICIENTLY, AND FAIRLY.
  - -- THERE SHOULD BE NO POSSIBILITY OF ONCOMING TRAINS COLLIDING.
  - -- THE SCHEME SHOULD NOT DEADLOCK.
    - IDEALLY, SOME ENGINEER SHOULD BE ABLE TO PROCEED WITHIN A PREDETERMINED TIME INTERVAL IF ANY ENGINEER WANTS TO USE THE SHARED TRACK.
    - AT THE VERY LEAST, THERE SHOULD BE SOME FIXED MINIMUM PROBABILITY THAT SOME ENGINEER WILL BE ALLOWED TO PROCEED WITHIN THE NEXT MINUTE WHENEVER ANY ENGINEER WANTS TO USE THE SHARED TRACK.
  - -- EASTBOUND AND WESTBOUND TRAINS SHOULD BE TREATED EQUALLY IN THE LONG RUN.
- THE ENGINEERS MAY COMMUNICATE AS FOLLOWS:
  - -- EACH ENGINEER HAS A SIGNAL FLAG THAT MAY BE RAISED OR LOWERED. IF ONE ENGINEER RAISES A SIGNAL FLAG, AN ENGINEER IN AN ONCOMING TRAIN CAN SEE IT.
  - -- THERE IS A BOULDER THAT MAY BE PICKED UP FROM ONE END OF THE SHARED TRACK AND DROPPED AT THE OTHER END. AN ENGINEER CAN SEE WHETHER THE BOULDER IS PRESENT AT HIS END.
- YOU MAY ASSUME THAT ALL ENGINEERS WORK AT DIFFERENT SPEEDS. IT IS EXTREMELY UNLIKELY THAT THEY WILL PERPETUALLY BE FOLLOWING EXACTLY THE SAME INSTRUCTIONS AT EXACTLY THE SAME TIME, SO YOU CAN IGNORE THIS POSSIBILITY.
- WRITE AN ADA PROGRAM SPECIFYING WHAT EACH ENGINEER SHOULD DO AS HE APPROACHES THE TRACK. YOU MAY USE ANY OR ALL OF THE FOLLOWING SUBPROGRAMS:

```
procedure Raise_Flag;                      function Boulder_Present
procedure Lower_Flag;                         return Boolean;
function Oncoming_Flag_Raised              procedure Pick_Up_Boulder;
   return Boolean;                         procedure Drop_Boulder;

procedure Cross_River; -- Throw the switches and use the shared track.
procedure Wait;        -- Do nothing for some fixed amount of time.
```

IF YOU KNOW ANY OF ADA'S MULTITASKING FEATURES, DO NOT USE THEM.

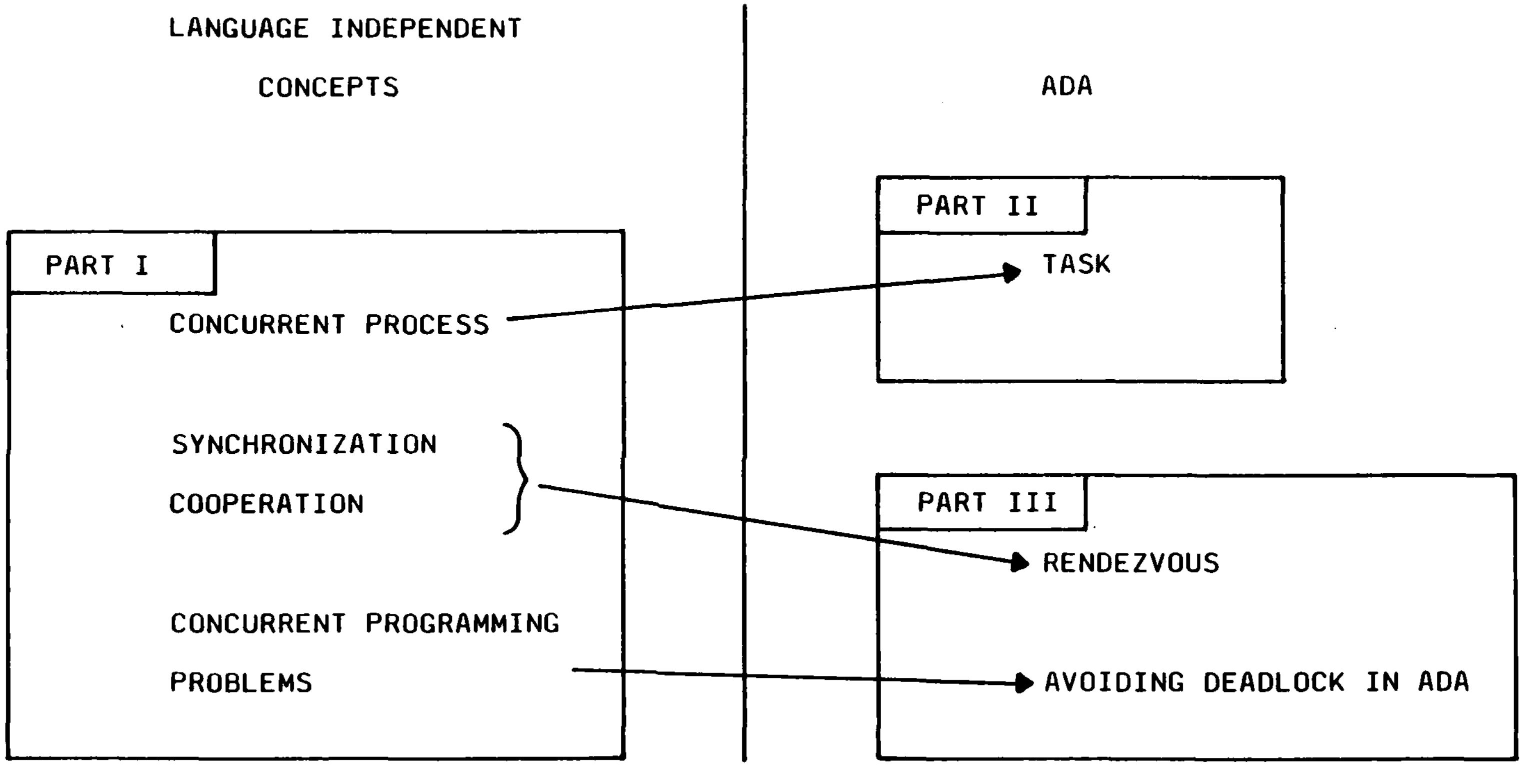
INSTRUCTOR NOTES

PRESENT THIS SLIDE BEFORE BEGINNING PART II. IT PROVIDES A TRANSITION BETWEEN PART I AND THE REST OF THE COURSE.

THE LEFT HALF OF THE SLIDE ILLUSTRATES LANGUAGE-INDEPENDENT CONCEPTS COVERED IN PART I. THE RIGHT HALF SHOWS THE CORRESPONDING FEATURES OF ADA.

AS WE MOVE FROM A DISCUSSION OF GENERAL PRINCIPLES TO A DISCUSSION OF ADA SPECIFIES, THE DISCUSSION WILL GET MORE TECHNICAL. WARN THE STUDENTS.

RELATIONSHIP BETWEEN PARTS I, II AND III



LANGUAGE INDEPENDENT CONCEPTS

ADA

PART I

CONCURRENT PROCESS

SYNCHRONIZATION

COOPERATION

CONCURRENT PROGRAMMING PROBLEMS

PART II

TASK

PART III

RENDEZVOUS

AVOIDING DEADLOCK IN ADA

INSTRUCTOR NOTES

PART II

ADA TASKING CONCEPTS

4. TASK TYPES AND TASK OBJECTS

5. TASK DECLARATIONS AND TASK TYPES

6. TASK ACTIVATION AND TERMINATION

INSTRUCTOR NOTES

- ALLOW 20 MINUTES FOR THIS SECTION.
- THIS SECTION INTRODUCES THE Ada CONCEPT OF A TASK AS A DATA OBJECT BELONGING TO A TASK TYPE.
- THE CONCEPTS IN THIS SECTION WILL BE NEW TO MANY STUDENTS, SO GO THROUGH THE SECTION SLOWLY.
- STARTING IN SECTION 5 WE WILL GIVE ACTUAL Ada CODE.
- RECOMMEND THAT STUDENTS READ EXERCISE 2.1 IN THE REAL-TIME ADA WORKBOOK.

Section 4

TASK TYPES AND TASK OBJECTS

INSTRUCTOR NOTES

- IN THIS SECTION, WE START TO INTRODUCE THE STUDENT TO THE Ada VIEW OF A PROCESS. THE VIEW THAT WE GIVE HERE IS THAT A TASK OBJECT IS A DATA OBJECT THAT BELONGS TO A TASK TYPE. IN THIS RESPECT, AN Ada TASK OBJECT IS NOT MUCH DIFFERENT THAN DATA OBJECTS WE TALKED ABOUT IN EARLIER COURSES. THIS IS THE MAIN POINT OF THIS SECTION AND IT WILL PERMEATE THE REST OF THE COURSE. EVERY ATTEMPT SHOULD BE MADE TO GET THIS POINT ACROSS IN THIS SECTION. HOWEVER, THIS MIGHT BE TOO TASKING (SORRY ABOUT THAT!) FOR MOST OF THE STUDENTS TO GRASP RIGHT AWAY.

- GO THROUGH THE SLIDES SLOWLY.

- IT IS IMPORTANT THAT THE TERM TASK OBJECT IS USED CONSISTENTLY IN THIS SECTION. STARTING IN THE NEXT SECTION, WE WILL START TO USE TASK INSTEAD OF TASK OBJECT.

WHAT IS A PROCESS IN Ada?

- IN Ada, PROCESSES ARE REALIZED AS TASK OBJECTS.
- EACH PROCESS IS EXECUTED BY ITS OWN VIRTUAL PROCESSOR.
- A TASK OBJECT IS A DATA OBJECT CONSISTING OF:
  - A PROCESS EXECUTING A PARTICULAR SEQUENCE OF STATEMENTS.
  - A SET OF DATA RESERVED FOR THE EXCLUSIVE USE OF THAT PROCESS.
  - ENTRIES THROUGH WHICH IT IS POSSIBLE TO COMMUNICATE WITH THIS PROCESS.
- LIKE ANY DATA OBJECT IN Ada, A TASK OBJECT BELONGS TO A TYPE - A TASK TYPE.

INSTRUCTOR NOTES

- THIS SLIDE IS A REVIEW THAT PREPARES THE STUDENTS FOR THE NEXT SLIDE.

- BULLET #1 - THIS GIVES TWO DIVERSE EXAMPLES.

REVIEW - TYPES AS SETS OF VALUES PLUS OPERATIONS

- EXAMPLES:
  - INTEGERS
    - VALUES CONSIST OF ALL INTEGERS (UP TO AN IMPLEMENTATION DEFINED LIMIT).
    - MANIPULATIONS INCLUDE ADDITION, SUBTRACTION, ETC.
  - AN UNCONSTRAINED CHARACTER ARRAY TYPE
    - VALUES ARE CONSTRAINED CHARACTER ARRAYS
    - MANIPULATIONS INCLUDE SELECTION, SLICING, ETC.
- VALUES IN THE SAME TYPE SHARE COMMON CHARACTERISTICS:
  - ALL VALUES IN THE TYPE HAVE THE SAME FORM.
  - ALL VALUES ARE SUBJECT TO THE SAME MANIPULATIONS.

INSTRUCTOR NOTES

- THIS SLIDE SHOWS THAT THE IDEAS ON THE PREVIOUS SLIDE ARE ALSO VALID FOR TASK TYPES.
- THE ETC. INCLUDES ATTRIBUTES (MENTION THIS) AND THE ABORT STATEMENT (DON'T MENTION THIS).

WHAT IS A TASK TYPE?

- LIKE ANY OTHER Ada TYPE, A TASK TYPE HAS:
  - A SET OF VALUES.
    - THE VALUES ARE TASK OBJECTS.
  - A SET OF WAYS TO MANIPULATE THESE VALUES.
    - STARTING A TASK OBJECT RUNNING.
    - COMMUNICATING WITH A TASK OBJECT THROUGH ONE OF ITS ENTRIES.
    - ETC.

- LIKE ANY OTHER Ada TYPE, TWO TASK OBJECTS BELONGING TO THE SAME TASK TYPE SHARE COMMON CHARACTERISTICS.
  - THEIR PROCESSES EXECUTE THE SAME SEQUENCE OF STATEMENTS.
  - THEIR DATA OBJECTS ARE DESCRIBED BY THE SAME DECLARATIONS.
  - THEIR ENTRIES HAVE THE SAME FORM.

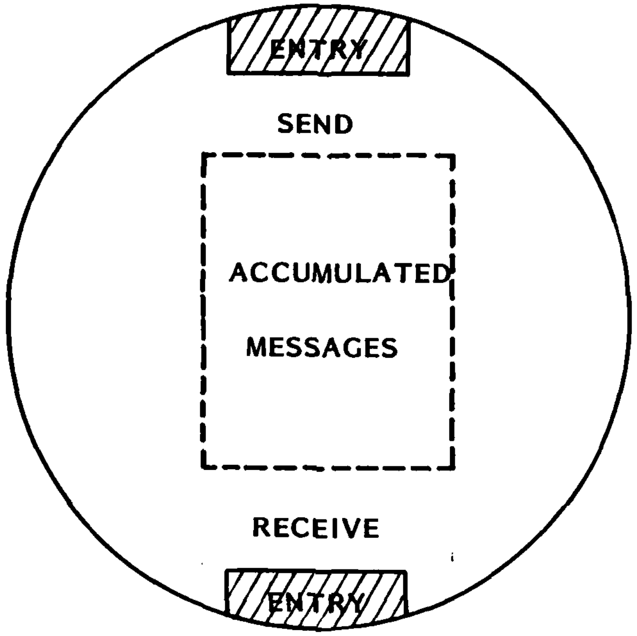
INSTRUCTOR NOTES

- SPEND ALMOST NO TIME ON WHAT A Message_Buffer_Type IS USED FOR. THE PURPOSE OF THIS SECTION IS TO MAKE THE STUDENTS UNDERSTAND THAT A TASK OBJECT IS A DATA OBJECT.

- FOR THE INSTRUCTOR'S BENEFIT, A Message_Buffer_Type TASK OBJECT PROVIDES THE ABILITY TO SEND MESSAGES FASTER THAN THEY CAN BE RECEIVED. WHY YOU WOULD WANT TO DO THIS WILL BE COVERED LATER IN THE COURSE.

- THE USER OF THE Message_Buffer_Type SHOULD BE NO MORE CONCERNED ABOUT THESE DETAILS THAN THE USER OF AN ABSTRACT DATA TYPE, SAY Varying_String_Type, IS ABOUT THE INTERNAL DETAILS OF THAT TYPE. THE USER SHOULD ONLY BE CONCERNED WITH THE OPERATIONS ON A Varying_String_Type. THIS POINT NEEDS TO BE EMPHASIZED!

EXAMPLE OF A TASK TYPE: Message_Buffer_Type



- OBJECTS OF TYPE Message_Buffer_Type ARE TASK OBJECTS.
- MANIPULATIONS OF A Message_Buffer_Type TASK OBJECT INCLUDE:
  - COMMUNICATING WITH THE TASK OBJECT THROUGH THE Send ENTRY.
  - COMMUNICATING WITH THE TASK OBJECT THROUGH THE Receive ENTRY.
  - STARTING EXECUTION OF THE TASK OBJECT.

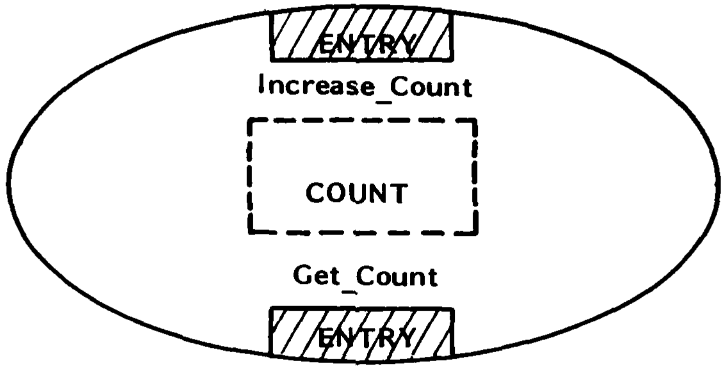
INSTRUCTOR NOTES

- AGAIN, SPEND ALMOST NO TIME ON WHAT A Shared_Count_Type IS USED FOR.

- A Shared_Count_Type ALLOWS TWO OR MORE PROCESSES TO COUNT THE NUMBER OF TIMES THAT SOME EVENT OF INTEREST OCCURS. (THIS IS AN EXAMPLE OF SHARED UPDATE THAT WILL BE RETURNED TO LATER IN THE COURSE.)

- TASK OBJECTS IN THIS TYPE MAINTAIN A SEPARATE COUNT.

EXAMPLE OF A TASK TYPE: Shared_Count_Type



- OBJECTS OF TYPE Shared_Count_Type ARE TASK OBJECTS.

- MANIPULATIONS OF A Shared_Count_Type TASK OBJECT INCLUDE:
  - COMMUNICATING WITH THE TASK OBJECT THROUGH THE Increase_Count ENTRY.
  - COMMUNICATING WITH THE TASK OBJECT THROUGH THE Get_Count ENTRY.
  - STARTING EXECUTION OF THE TASK OBJECT.

INSTRUCTOR NOTES

- ALLOW 60 MINUTES FOR THIS SECTION.

- THIS WILL BE FOLLOWED BY A 30 MINUTE EXERCISE AT THE END.

- IN THIS SECTION WE START WITH THE SYNTAX OF TASK TYPE DECLARATIONS AND TASK BODIES. WE ALSO GIVE SOME SMALL ISOLATED EXAMPLES.

- THIS SECTION WILL ALSO SHOW HOW TO DECLARE TASK OBJECTS, ARRAYS OF TASKS, AND ACCESS TYPES DESIGNATING TASKS.

- ALTHOUGH WE WILL SHOW A FEW EXAMPLES OF ENTRY CALLS, WE WILL NOT TALK ABOUT ACCEPT STATEMENTS UNTIL THE NEXT SECTION. ENTRY CALL EXAMPLES IN THIS SECTION ARE INCLUDED ONLY TO SHOW THE SYNTAX OF A CALL AND TO ADD SOME CONCRETENESS TO THIS SECTION. (I HOPE THIS WON'T MAKE IT HARD -- SORRY AGAIN!)

- NOTE THAT IN THIS SECTION WE START TALKING ABOUT TASKS RATHER THAN TASK OBJECTS. DO NOT EXPLICITLY MENTION THIS TO THE CLASS, BUT IF ANYONE SHOULD ASK, THIS IS NO DIFFERENT THAN REFERRING TO AN ARRAY TYPE OBJECT AS AN ARRAY.

Section 5

TASK DECLARATIONS AND TASK TYPES

INSTRUCTOR NOTES

- THIS SLIDE TALKS ABOUT SOME OF THE SAME THINGS THE PREVIOUS SECTION DID, BUT IN THE CONTEXT OF TASK TYPE DECLARATIONS AND TASK BODIES. THE NEXT SLIDE GIVES THE SYNTAX.

DECLARING A TASK TYPE

- THE DEFINITION OF A TASK TYPE HAS TWO PARTS:
  - A TASK TYPE DECLARATION
    - DECLARES THE TASK TYPE.
    - DECLARES THE FORM OF ENTRIES FOR TASKS IN THE TASK TYPE.
  - A TASK BODY
    - DEFINES A TEMPLATE FOR DATA USED BY EACH TASK IN THE TYPE.
    - DEFINES THE SEQUENCE OF STATEMENTS TO BE EXECUTED BY A TASK IN THE TYPE.

- EACH TASK IN THE TASK TYPE
  - HAS ENTRIES OF THE FORM DESCRIBED IN THE TASK TYPE DECLARATION.
  - HAS ITS OWN PRIVATE COPY OF THE DATA DESCRIBED BY THE TEMPLATE IN THE TASK BODY.
  - EXECUTES THE SEQUENCE OF STATEMENTS IN THE TASK BODY (AT A SPEED INDEPENDENT OF ANY OTHER TASK, INCLUDING TASKS OF THE SAME TYPE).

- THE TASK TYPE DECLARATION AND TASK BODY TOGETHER ARE CALLED A TASK UNIT.

INSTRUCTOR NOTES

- THIS SLIDE INTRODUCES THE SYNTAX OF A BASIC TASK TYPE DECLARATION. TASKS WITHOUT ENTRIES WILL BE DISCUSSED AT THE END OF THE SECTION, AS WILL ONE OF A KIND TASKS. ENTRY FAMILIES WILL BE DISCUSSED LATER IN THE COURSE.

SYNTAX OF A BASIC TASK TYPE DECLARATION

```
task type task type name is

   entry entry name [( formal parameters )];

end [ task type name ];
```

- task type name IS AN IDENTIFIER FOR NAMING THE TYPE. THE SECOND OCCURRENCE OF THIS IDENTIFIER IS OPTIONAL, BUT GOOD PROGRAMMING STYLE SUGGESTS THAT IT NEVER BE OMITTED.

- THE FIRST AND LAST LINES SURROUND ZERO OR MORE ENTRY DECLARATIONS.

  - AN ENTRY IS THE MEANS BY WHICH OTHER TASKS COMMUNICATE WITH A TASK IN THIS TYPE.

  - ENTRY DECLARATIONS ARE LIKE PROCEDURE DECLARATIONS WITH THE WORD procedure REPLACED BY THE WORD entry.

    - PARAMETER MODES CAN BE in, in out, or out

    - DEFAULT INITIAL VALUES CAN BE SPECIFIED FOR in PARAMETERS.

INSTRUCTOR NOTES

- GO OVER EACH EXAMPLE IN ENOUGH DETAIL TO MAKE THE SYNTAX CLEAR.
- THESE EXAMPLES WERE INTRODUCED IN THE PREVIOUS SECTION. WE ARE JUST GOING TO FLESH THEM OUT A LITTLE.
- THE PURPOSE OF THIS SLIDE IS TO ILLUSTRATE TASK TYPE DECLARATIONS.
- EMPHASIZE HOW MUCH THE ENTRY DECLARATIONS LOOK LIKE PROCEDURE DECLARATIONS. THESE ARE USER DEFINED OPERATIONS ON TASK TYPE OBJECTS MUCH LIKE SUBPROGRAMS DEFINED IN A PACKAGE DECLARATION ARE OPERATIONS ON THE TYPES DECLARED IN THE PACKAGE DECLARATION.

EXAMPLES OF BASIC TASK TYPE DECLARATIONS

CONTEXT FOR EXAMPLE:

```
type Message_Type is ...;
```

EXAMPLE:

```
task type Message_Buffer_Type is

   entry Send (Message:  in Message_Type);
   entry Receive (Message:  out Message_Type);

end Message_Buffer_Type;
```

EXAMPLE:

```
task type Shared_Count_Type is

   entry Increase_Count (By:  in Positive);
   entry Get_Count (Sum_So_Far:  out Natural);

end Shared_Count_Type;
```

INSTRUCTOR NOTES

- THIS SLIDE INTRODUCES THE SYNTAX OF A TASK BODY.

- FIRST BULLET - STRESS THIS SIMILARITY.

- LAST BULLET - THESE ARE THE ACCEPT STATEMENT, SELECTIVE WAIT, ETC. THEY WILL BE COVERED IN DETAIL IN SECTION 7.

SYNTAX OF A TASK BODY

```
task body task type name is
   declarative part
begin
   sequence of statements
    [exception
       sequence of exception handlers]
end [task type name];
```

- THE FORM OF A TASK BODY IS SIMILAR TO THAT OF A SUBPROGRAM BODY, A PACKAGE BODY AND A BLOCK STATEMENT. THE DIFFERENCE IS THE HEADER LINE.

- task type name IS AN IDENTIFIER NAMING THE TASK TYPE FOR WHICH THIS IS THE TASK BODY. WHILE THE SECOND OCCURRENCE OF THE IDENTIFIER IS OPTIONAL, GOOD PROGRAMMING STYLE SUGGESTS THAT IT NEVER BE OMITTED.

- declarative part ACTS AS A TEMPLATE FOR DATA USED BY EACH TASK IN THE TASK TYPE.

- sequence of statements DEFINES THE SEQUENCE OF STATEMENTS THAT A TASK IN THIS TASK TYPE WILL EXECUTE.

- THE TASK BODY MAY CONTAIN CERTAIN OTHER STATEMENTS NOT ALLOWED ELSEWHERE IN Ada PROGRAMS.

INSTRUCTOR NOTES

- THE DECLARATIONS AND STATEMENTS FOR THE Message_Buffer_Type TASK WILL BE PROVIDED LATER IN THE COURSE.

- THE STATEMENTS FOR THE Shared_Count_Type TASK WILL BE PROVIDED IN SECTION 7.

- IN THE Shared_Count_Type EACH TASK IN THIS TYPE GETS ITS OWN PRIVATE COPY OF Sum, INITIALIZED TO ZERO.

EXAMPLES OF TASK BODIES

```
task body Message_Buffer_Type is

      -- DECLARATIONS GO HERE

begin -- Message_Buffer_Type

      -- SEQUENCE OF STATEMENTS GOES HERE

end Message_Buffer_Type;



task body Shared_Count_Type is

      Sum:  Natural := 0;

begin -- Shared_Count_Type

      -- SEQUENCE OF STATEMENTS GOES HERE

end Shared_Count_Type;
```

INSTRUCTOR NOTES

- BULLET 1 - THE TWO ITEMS HOLD FOR ANY LIMITED TYPE.

- REMEMBER THAT A RECORD TYPE THAT HAS A TASK TYPE COMPONENT IS LIMITED AS A CONSEQUENCE OF TASK TYPES BEING LIMITED. ASSIGNMENT AND (PREDEFINED) EQUALITY DO NOT EXIST. THE SAME IS TRUE FOR AN ARRAY TYPE OR RECORD TYPE WITH A COMPONENT HAVING A TASK TYPE.

- FOR THE SECOND BULLET, DO NOT TRY TO EXPLAIN WHAT MODE *in out* IS USED FOR. HOWEVER, JUST IN CASE YOU ARE PRESSED AND CAN'T GET OUT OF IT, IT SIMPLY SERVES AS DOCUMENTATION. ITS EFFECT IS NO DIFFERENT THAN IF YOU USED MODE *in*. HOWEVER, IF WITHIN THE SUBPROGRAM BODY ONE OF THE TASK'S ENTRIES IS CALLED THAT MODIFIES THE TASK'S PRIVATE DATA, THEN MODE *in out* MIGHT BE USED TO DOCUMENT THIS.

USE OF TASK TYPES

- TASK TYPES ARE LIMITED TYPES.
  - VALUES IN THE TYPE CANNOT BE COMPARED (=, /=);
  - VALUES IN THE TYPE CANNOT BE COPIED.

- OTHER USES:
  - AS THE FULL TYPE DECLARATION IN A PACKAGE PRIVATE PART CORRESPONDING TO A LIMITED PRIVATE TYPE DECLARATION IN THE PACKAGE VISIBLE PART.
  - A SUBPROGRAM PARAMETER OF MODE in OR in out MAY BELONG TO A TASK TYPE. HOWEVER MODE out IS NOT ALLOWED.
  - A TASK TYPE CAN BE USED AS A GENERIC ACTUAL TYPE PARAMETER WHEN THE CORRESPONDING GENERIC FORMAL TYPE PARAMETER IS LIMITED.
  - A TASK OBJECT CAN BE USED AS A GENERIC ACTUAL OBJECT PARAMETER OF MODE in out WHEN THE TYPE OF THE GENERIC FORMAL OBJECT PARAMETER IS LIMITED PRIVATE AND HAS MODE in out.

- A TASK TYPE NAME MAY BE USED ANYWHERE AFTER THE TASK TYPE DECLARATION, I.E., THE TASK BODY NEED NOT BE SPECIFIED BEFORE THE TASK TYPE NAME IS USED.

INSTRUCTOR NOTES

- THE EXAMPLES IN THIS AND THE NEXT FEW SLIDES ARE PRIMARILY CONCERNED WITH HAMMERING IN THE POINT THAT TASK TYPES CAN BE USED LIKE OTHER Ada TYPES. DO NOT SPEND TOO MUCH TIME ON THESE EXAMPLES OTHER THAN TO MAKE ANY ADDITIONAL POINTS THAT A SLIDE MAY BE MAKING.

- THIS SLIDE SHOWS SIMPLE TASK DECLARATIONS FOR TWO TASK TYPES. THE ONE FOR THE Shared_Count_Type ILLUSTRATES TWO TASKS BEING DECLARED FOR A TYPE.

- WE ARE USING ENTRY CALLS IN THESE EXAMPLES EVEN THOUGH WE HAVE NOT FORMALLY INTRODUCED THEM. BY THIS STAGE OF THE GAME, STUDENTS SHOULD HAVE ENOUGH Ada MATURITY TO HANDLE THIS. YOU SHOULD MENTION TO THE STUDENTS THAT WE ARE DOING THIS, AND THAT FOR THIS SECTION THEY CAN THINK OF THEM AS PROCEDURE CALLS. ENTRY CALLS WILL BE TREATED FORMALLY IN SECTION 7.

- IN THE Shared_Count_Type, MENTION THE ADDED READABILITY OBTAINED BY USING NAMED NOTATION WHEN THE NAMES ARE INTELLIGENTLY CHOSEN.

- AS BACKGROUND FOR THE SECOND EXAMPLE, EACH ROOM IN A NUCLEAR POWER PLANT MIGHT HAVE A GEIGER COUNTER TO MONITOR RADIATION LEVELS. AS THE SENSORS OF THE GEIGER COUNTER SENSE RADIATION, THEY CALL THE Increase_Count ENTRY OF A Shared_Count_Type OBJECT. THIS ALLOWS THE GEIGER COUNTER TO KEEP TRACK OF THE NUMBER OF GEIGERS COUNTED SO FAR. WHEN TOO MANY GEIGERS ARE COUNTED AN ALARM WOULD BE SOUNDED.

- WARNING: THE PREVIOUS EXAMPLE IS Tongue-In Cheek. GEIGER COUNTERS DO NOT REALLY COUNT GEIGERS.

SIMPLE TASK DECLARATIONS

CONTEXT FOR EXAMPLE:

```
      Message:  Message_Type;
```

EXAMPLE:

```
      Message_Buffer:  Message_Buffer_Type;
             ...
      -- Get a message from the buffer.
      Message_Buffer.Receive (Message);
      -- Send a message to the buffer.
      Message_Buffer.Send (Message);
```

CONTEXT FOR EXAMPLE:

```
      Number_of_Geigers_Counted:  Positive;
```

EXAMPLE:

```
      Control_Room_Geiger_Counter, Locker_Room_Geiger_Counter:  Shared_Count_Type;

             -- The above declaration is equivalent to the declarations:
             --    Control_Room_Geiger_Counter:  Shared_Count_Type;
             --    Locker_Room_Geiger_Counter:   Shared_Count_Type;
             ...
      Control_Room_Geiger_Counter.Increase_Count (By => 1);
             ...
      Locker_Room_Geiger_Counter.Get_Count
                                        (Sum_So_Far =>  Number_of_Geigers_Counted);
```

INSTRUCTOR NOTES

- THIS SLIDE ILLUSTRATES THAT A TASK TYPE CAN BE USED AS THE COMPONENT TYPE OF AN ARRAY TYPE. MENTION AGAIN THAT THE ARRAY TYPE IS LIMITED.

- EXPLAIN THE SYNTAX OF THE ENTRY CALLS SINCE WE NOW HAVE AN ARRAY INDEX INVOLVED.

- MENTION THAT WE CAN HAVE MULTI-DIMENSIONAL ARRAYS OF TASKS OR ARRAYS WHOSE COMPONENTS ARE ARRAYS OF TASKS, ETC. THIS IS TRUE OF ANY COMPONENT TYPE.

- IN THIS EXAMPLE, WE HAVE A DISCRETE TYPE NAMING THE AREAS THAT ARE TO BE MONITORED.

ARRAYS OF TASKS

CONTEXT FOR EXAMPLE:

```
type Monitored_Areas_Type is (Control_Room, Locker_Room, ...);
```

EXAMPLE:

```
type Geiger_Counter_List_Type is array (Monitored_Areas_Type) of Shared_Count_Type;
   -- Geiger_Counter_List_Type IS A LIMITED TYPE
Geiger_Counter_List:  Geiger_Counter_List_Type;
             ...


Geiger_Counter_List (Control_Room).Increase_Count (By => 1);
Geiger_Counter_List (Locker_Room).Increase_Count (By => Number_of_Geigers_Counted);
```

INSTRUCTOR NOTES

- THIS SLIDE ILLUSTRATES THAT A TASK TYPE CAN BE USED AS A COMPONENT TYPE OF A RECORD TYPE. MENTION AGAIN THAT THE RECORD TYPE IS LIMITED.

- EXPLAIN THE SYNTAX OF THE ENTRY CALLS SINCE WE NOW HAVE A SELECTED RECORD COMPONENT.

- MENTION THAT WE COULD HAVE COMPONENTS THAT ARE ARRAYS OF TASKS OR RECORDS WITH TASK COMPONENTS. ALSO WE CAN HAVE ARRAYS WHOSE COMPONENTS ARE RECORDS WITH TASK COMPONENTS. AGAIN THIS IS NO DIFFERENT THAN FOR OTHER TYPES.

- IN THIS EXAMPLE, WE HAVE NAMES ASSOCIATED WITH EACH COUNTER. THE PROCEDURE SIMPLY PRINTS THE NAME OF THE COUNTER AND ITS CURRENT VALUE.

TASK TYPES AS RECORD COMPONENTS

- CONTEXT:

```
type Named_Counter_Type is
   record
       Name_Part    : String (1 .. 20);
       Counter_Part : Shared_Count_Type;
   end record;
       ...
```

- EXAMPLE:

```
with Text_IO; use Text_IO;

procedure Print_Counter_Data (Named_Counter : in Named_Counter_Type) is

   Current_Count : Positive;

   package Type_Integer_IO is new Integer_IO (Integer);
   use Type_Integer_IO;

begin

   Put ("Current value of counter");
   Put (Named_Counter.Name_Part);
   Put ("is");
   Named_Counter.Counter_Part.Get_Count (Current_Count);
   Put (Current_Count);
   Put (".");
   New_Line;

end Print_Counter_Data;
```

INSTRUCTOR NOTES

- THIS SLIDE ILLUSTRATES THE USE OF TASK TYPES AS THE TYPE OF OBJECT DESIGNATED BY AN ACCESS TYPE.

- EVEN THOUGH TWO TASKS CANNOT BE COMPARED FOR EQUALITY, A PROGRAM NEEDING TO DETERMINE WHETHER TWO TASKS ARE THE SAME CAN USE ACCESS TYPES TO ACCOMPLISH THIS. (AN ACCESS TYPE DESIGNATING A LIMITED TYPE IS NOT LIMITED.) IF TWO ACCESS VALUES ARE THE SAME, THEN THEY DESIGNATE THE SAME TASK.

ACCESS TYPES AND TASK TYPES

CONTEXT FOR EXAMPLE:

```
type Message_Buffer_Pointer_Type is access Message_Buffer_Type;
       ...
```

EXAMPLE:

```
Operator_Console_Message_Buffer : Message_Buffer_Pointer_Type :=
                                      new Message_Buffer_Type;
       ...
procedure Log_Hard_Copy (Of_Message : Message_Type) is
       ...
end Log_Hard_Copy;
       ...
procedure Send (Message        : in Message_Type;
                Message_Buffer : in Message_Buffer_Pointer_Type) is
       ...
begin
       ...

   if Message_Buffer = Operator_Console_Message_Buffer then

       Log_Hard_Copy (Of_Message => Message);

   end if;
   Message_Buffer.Send (Message);
       ...
end Send;
```

INSTRUCTOR NOTES

- BULLET #1

    - THE PLACES A TASK TYPE DECLARATION CAN APPEAR ARE ESSENTIALLY THE SAME PLACES WHERE ANY TYPE DECLARATION CAN APPEAR.

    - THE ONLY ADDITION IS THE ALLOWANCE OF A TASK TYPE DECLARATION AFTER A TASK BODY. THIS ALLOWS SUBSEQUENT TASK TYPE DECLARATIONS TO APPEAR JUST BEFORE THEIR TASK BODIES.

    - A TASK UNIT IS PLACED IN THE DECLARATIVE PART OF A PACKAGE WHEN IT IS TO BE USED IN IMPLEMENTING THE PACKAGE. IT IS NOT PART OF THE INTERFACE (SINCE IT IS NOT IN THE VISIBLE PART OF THE PACKAGE DECLARATION).

- EXAMPLES WILL APPEAR IN THE NEXT FEW SLIDES

WHERE DO TASK UNITS GO?

- IN THE DECLARATIVE PART OF A BLOCK STATEMENT, A SUBPROGRAM BODY, A PACKAGE BODY, OR AN OUTER TASK BODY.
  - THE TASK TYPE DECLARATION APPEARS FIRST.
  - THE TASK BODY GOES SOMEWHERE AFTER THE TASK DECLARATION. JUST AS FOR SUBPROGRAM BODIES AND PACKAGE BODIES, THE FOLLOWING MAY NOT APPEAR AFTER THE BODY:
    - DECLARATIONS OF OBJECTS AND NAMED NUMBERS
    - TYPE DECLARATIONS OTHER THAN TASK TYPE DECLARATIONS
    - SUBTYPE DECLARATIONS
    - EXCEPTION DECLARATIONS
    - RENAMING DECLARATIONS
- IF THE TASK TYPE IS PROVIDED BY A PACKAGE (AS PART OF THE PACKAGE INTERFACE) THEN
  - THE TASK TYPE DECLARATION GOES IN THE PACKAGE DECLARATION.
  - THE TASK BODY GOES IN THE DECLARATIVE PART OF THE PACKAGE BODY.
- TASK BODIES CAN BE REPLACED BY BODY STUBS AND COMPILED SEPARATELY AS SUBUNITS:

```
task body | task type name | is separate;
```

INSTRUCTOR NOTES

- THIS SLIDE HAS TWO MAIN POINTS TO GET ACROSS.

  1. TASK UNITS CANNOT BE COMPILED SEPARATELY AS LIBRARY UNITS. (REMIND THE CLASS THAT A LIBRARY UNIT IS A SEPARATELY COMPILED UNIT MADE AVAILABLE WITH A with CLAUSE.) HOWEVER THIS DOES NOT RESULT IN ANY LOSS OF FUNCTIONALITY AS THE EXAMPLE ILLUSTRATES.

  2. WHEN A TASK TYPE IS PLACED IN THE VISIBLE PART OF A PACKAGE DECLARATION THEN THE TASK TYPE IS AVAILABLE TO USERS OF THE PACKAGE. NOTE THE USE OF THE PACKAGE NAME IN QUALIFYING THE Shared_Count_Type OBJECT. HOWEVER, IN USING THE Increase_Count ENTRY, THE PACKAGE NAME WAS NOT NEEDED.

- IF STUDENTS ASK WHY TASK UNITS CANNOT BE SEPARATELY COMPILED, POINT OUT THAT A TASK UNIT IS IN SOME SENSE A HYBRID. IT CONSISTS OF A TASK TYPE DECLARATION AND A TASK BODY. IN THIS WAY IT MAKES SENSE - WHAT DOES IT MEAN TO SEPARATELY COMPILE A TYPE? IF PRESSED MENTION THAT THERE IS ALSO A SMALL TECHNICAL PROBLEM (DEALING WITH MASTERS).

TASK UNITS AND PACKAGES

- TASK TYPE DECLARATIONS AND TASK BODIES MAY NOT BE COMPILED SEPARATELY AS LIBRARY UNITS. THE EXAMPLE SHOWS HOW TO ACHIEVE THE SAME EFFECT.

```
package Shared_Count_Package is

   task type Shared_Count_Type is
      ...
   end Shared_Count_Type;

end Shared_Count_Package;


package body Shared_Count_Package is

   task body Shared_Count_Type is
      ...
   end Shared_Count_Type;

end Shared_Count_Package;
```

USING THE PACKAGE:

```
with Shared_Count_Package;

procedure Count_Geigers is
      ...
   Control_Room_Geiger_Counter : Shared_Count_Package.Shared_Count_Type;
      ...
begin
      ...
   Control_Room_Geiger_Counter.Increase_Count (By => 1);
      ...
end Count_Geigers;
```

INSTRUCTOR NOTES

- THIS EXAMPLE SHOWS HOW A TASK TYPE CAN BE USED AS THE FULL TYPE DECLARATION FOR A LIMITED PRIVATE TYPE. NOTE THAT THE TYPE CANNOT BE JUST PRIVATE, IT MUST BE LIMITED PRIVATE SINCE A TASK TYPE IS A LIMITED TYPE.

- NOTE THE USE OF THE RENAMING OF By WITHIN PROCEDURE Update IN ORDER TO IMPROVE READABILITY.

- NOTE AGAIN THE USE OF TASKS AS PARAMETERS.

- Geiger_Counter_Type IS THE SAME AS THE Shared_Count_Type USED EARLIER.

TASK TYPES AS LIMITED PRIVATE TYPES

```
package Geiger_Package is

   type Geiger_Counter_Type is limited private;

   procedure Update (Geiger_Of : in Geiger_Counter_Type; By : in Positive);
   procedure Get (Geiger_Of : in Geiger_Counter_Type; Sum_So_Far : out Natural);

private

   task type Geiger_Counter_Type is
      ...
   end Geiger_Counter_Type;

end Geiger_Package;

package body Geiger_Package is

   task body Geiger_Counter_Type is
      ...
   end Geiger_Counter_Type;

   procedure Update (Geiger_Of : in Geiger_Counter_Type; By : in Positive) is
      Number_Of_Geigers_Counted : Positive renames By;
   begin
      Geiger_Of.Increase_Count (By => Number_Of_Geigers_Counted);
   end Update;

   procedure Get (Geiger_Of : in Geiger_Counter_Type; Sum_So_Far : out Natural) is
      ...
   end Get;

end Geiger_Package;
```

INSTRUCTOR NOTES

- NOTE THAT THE USER OF THE PACKAGE IS NOT AWARE OF THE FACT THAT A TASK TYPE IS BEING USED TO IMPLEMENT Geiger_Counter_Type.

TASK TYPES AS LIMITED PRIVATE TYPES - CONTINUED

USING THE PACKAGE

```
with Geiger_Package;
procedure Count_Geigers;
       ...
   Control_Room_Geiger_Counter : Geiger_Package.Geiger_Counter_Type;
       ...
begin
       ...
   Geiger_Package.Update (Control_Room_Geiger_Counter, By => 1);
       ...
end Count_Geigers;
```

INSTRUCTOR NOTES

- THIS EXAMPLE ILLUSTRATES HOW YOU ACHIEVE THE EFFECT OF GENERIC TASK TYPES.

TASK TYPES AND GENERIC UNITS

- THERE ARE NO GENERIC TASK UNITS. THE EXAMPLE SHOWS HOW TO ACHIEVE THE SAME EFFECT.

```
generic

   type Message_Type is private;

package Message_Buffer_Template is

   task type Message_Buffer_Type is
       entry Send (Message : in Message_Type);
       entry Receive (Message : out Message_Type);
   end Message_Buffer_Type;

end Message_Buffer_Template;


package body Message_Buffer_Template is

   task body Message_Buffer_Type is
       ...
   end Message_Buffer_Type;

end Message_Buffer_Template;
```

INSTRUCTOR NOTES

- NOTE THE USE OF THE SUBTYPE DECLARATION TO ACHIEVE A RENAMING.
- NOTE THE USE OF THE RENAMING DECLARATIONS TO IMPROVE READABILITY AND TO PROVIDE MORE MEANINGFUL NAMES.

TASK TYPES AND GENERIC UNITS - CONTINUED

USING THE GENERIC PACKAGE:

```
with Message_Buffer_Template;
       ...

package Text_Message_Buffer_Package is
   new Message_Buffer_Template (Message_Type => Text_Message_Type);

subtype Text_Message_Buffer_Type is
       Text_Message_Buffer_Package.Message_Type;

procedure Send_Text_Message (Message : in Text_Message_Type)
       renames Text_Message_Buffer_Package.Send;

procedure Receive_Text_Message (Message : out Text_Message_Type)
       renames Text_Message_Buffer_Package.Receive;

       ...
```

INSTRUCTOR NOTES

- THIS SLIDE INTRODUCES TASK UNITS AS PROGRAM UNITS. TOGETHER WITH THE NEXT SLIDE, WE WANT TO SHOW THE STUDENTS THAT TASK UNITS FIT INTO THE LANGUAGE IN A MANNER THAT IS CONSISTENT WITH OTHER PROGRAM UNITS.

- AS INDICATED IN A EARLIER NOTE, TASK UNITS ARE IN SOME WAYS A HYBRID CONSISTING OF A TASK TYPE DECLARATION AND A TASK BODY. HERE WE WANT TO EMPHASIZE THE FACT THAT THEY ALSO BEHAVE PRETTY MUCH LIKE OTHER PROGRAM UNITS.

PROGRAM UNITS

- Ada PROVIDES FOUR KINDS OF PROGRAM UNITS:

  1. SUBPROGRAMS
     - PROCEDURES
     - FUNCTIONS

  2. PACKAGES

  3. GENERIC UNITS
     - GENERIC SUBPROGRAMS
     - GENERIC PACKAGES

  4. TASK UNITS

INSTRUCTOR'S NOTES

- BULLET # 1
  - SUB-BULLET #1
    - REMEMBER THAT IN CERTAIN CONTEXTS, SUBPROGRAM DECLARATIONS MAY BE OMITTED, AND SUBPROGRAM BODIES WILL SERVE BOTH ROLES. THIS IS NOT VALID FOR THE OTHER PROGRAM UNITS.
  - SUB-BULLET #4
    - A CALL ON A SUBPROGRAM MAY APPEAR AFTER THE SUBPROGRAM DECLARATION AND BEFORE THE BODY.
    - AN ENTITY PROVIDED BY A PACKAGE MAY BE REFERRED TO AFTER THE PACKAGE DECLARATION AND BEFORE THE PACKAGE BODY.
    - A GENERIC UNIT MAY BE INSTANTIATED AFTER THE GENERIC DECLARATION AND BEFORE THE GENERIC BODY.
    - A TASK TYPE MAY BE NAMED IN OBJECT DECLARATIONS, TYPE DECLARATIONS ETC., AFTER THE TASK TYPE DECLARATION AND BEFORE THE TASK BODY. SIMILARLY, ENTRY CALLS FOR OBJECTS OF THE TYPE MAY APPEAR AFTER THE TASK TYPE DECLARATION AND BEFORE THE TASK BODY.

PROGRAM UNITS - CONTINUED

	SUBPROGRAMS	PACKAGES	GENERIC UNITS	TASK UNITS
EXTERNAL VIEW	SUBPROGRAM DECLARATION	PACKAGE DECLARATION	GENERIC SUBPROGRAM OR GENERIC PACKAGE DECLARATION	TASK TYPE DECLARATION
IMPLEMENTATION	SUBPROGRAM BODY	PACKAGE BODY	SUBPROGRAM OR PACKAGE BODY	TASK BODY

- SIMILARITIES AMONG PROGRAM UNITS
  - EACH HAS TWO PARTS: A DECLARATION DESCRIBING THE EXTERNAL VIEW AND A BODY DESCRIBING THE IMPLEMENTATION.
  - THE EXTERNAL VIEW AND BODY MAY BOTH APPEAR IN A DECLARATIVE PART.
  - THE EXTERNAL VIEW CAN BE GIVEN IN A PACKAGE DECLARATION WITH THE INTERNAL VIEW IN A PACKAGE BODY.
  - EXTERNAL VIEW IS SUFFICIENT TO ALLOW USE OF THE UNIT.

INSTRUCTOR NOTES

- MAKE SURE THE CLASS REALIZES THAT THE DIFFERENCES DO NOT RESULT IN ANY REAL LIMITATIONS. THE EXAMPLES THEY JUST WENT THROUGH ILLUSTRATES THIS.

DIFFERENCES BETWEEN TASK UNITS AND OTHER PROGRAM UNITS

- SEPARATE COMPILATION
  - FOR SUBPROGRAMS, PACKAGES AND GENERIC UNITS:
    - THE EXTERNAL VIEW MAY BE COMPILED SEPARATELY AS A LIBRARY UNIT (TO BE MADE AVAILABLE THROUGH A <u>with</u> CLAUSE), AND
    - THE INTERNAL VIEW CAN BE COMPILED LATER AS A SECONDARY UNIT.
  - SEPARATE COMPILATION OF TASK UNITS IS ACCOMPLISHED BY ENCLOSING THE TASK UNIT IN A PACKAGE.

- GENERIC UNITS
  - THERE ARE GENERIC SUBPROGRAMS AND GENERIC PACKAGES.
  - THE EFFECT OF A "GENERIC TASK" IS ACHIEVED BY ENCLOSING A TASK UNIT IN A GENERIC PACKAGE.

INSTRUCTOR NOTES

- THIS SLIDE SERVES AS PREPARATION FOR THE NEXT SLIDE ON ANONYMOUS TASKS, BY REMINDING THE STUDENTS THAT THEY HAVE SEEN ANONYMOUS TYPES BEFORE.
- REMIND STUDENTS THAT WE OCCASIONALLY WANT TO DEFINE ONE-OF-A-KIND ARRAYS.

REVIEW - ANONYMOUS ARRAY TYPES

- Ada PROVIDES A SHORTHAND FOR DECLARING "ONE-OF-A-KIND" ARRAYS.

- THE DECLARATIONS

```
type Days_In_The_Month_Type is array (Month_Type) of Positive;
Days_In_Month : Days_In_The_Month_Type;
```

  CAN BE ABBREVIATED BY A SPECIAL KIND OF OBJECT DECLARATION:

```
Days_In_Month : array (Month_Type) of Positive;
```

- THE OBJECT Days_In_Month IS SAID TO BELONG TO AN ANONYMOUS ARRAY TYPE

INSTRUCTOR NOTES

- JUST AS WE SOMETIMES WANT TO DEFINE ONE-OF-A-KIND ARRAYS, WE SOMETIMES WANT TO DEFINE ONE-OF-A-KIND TASKS. AS AN EXAMPLE, IF WE ONLY WANTED ONE MESSAGE BUFFER FOR MESSAGES OF TYPE Message_Type THEN THE SLIDE SHOWS HOW TO DECLARE SUCH A TASK.

- THE SECOND VERSION IS IDENTICAL TO THE FIRST EXCEPT THAT THE RESERVED WORD type HAS BEEN OMITTED. THIS IS THE ONLY SYNTACTIC DIFFERENCE FROM THE FIRST ONE. OF COURSE, THE IDENTIFIER Message_Buffer IS THE NAME OF A TASK OBJECT, NOT A TASK TYPE. Ada TREATS THIS TASK AS BELONGING TO AN ANONYMOUS TASK TYPE JUST AS IN THE CASE OF ANONYMOUS ARRAYS. IN CASE A STUDENT ASKS HOW THIS IS DONE, JUST SAY THAT IT IS TREATED AS IF THE FOLLOWING DECLARATION EXISTED:

```
task type Anonymous_Type_For_Message_Buffer is ...;
Message_Buffer:  Anonymous_Type_For_Message_Buffer;
```

ANONYMOUS TASK TYPES

- THE SEQUENCE OF DECLARATIONS:

```
task type Message_Buffer_Type is  -- Message_Buffer_Type IS THE NAME OF A TASK TYPE
        ...
end Message_Buffer_Type;

task body Message_Buffer_Type is
        ...
end Message_Buffer_Type;

Message_Buffer: Message_Buffer_Type; -- Message_Buffer IS THE NAME OF A TASK OBJECT
```

CAN BE ABBREVIATED AS:

```
task Message_Buffer is -- THE WORD type IS MISSING AFTER task
        ...
end Message_Buffer;    -- Message_Buffer IS THE NAME OF A SINGLE TASK OBJECT
                       -- Its TYPE HAS NO NAME

task body Message_Buffer is
        ...
end Message_Buffer;
```

INSTRUCTOR NOTES

- SOMETIMES WE WANT TO HAVE A TASK THAT INITIATES COMMUNICATION WITH OTHER TASKS, BUT DOES NOT ITSELF NEED TO HAVE OTHER TASKS INITIATE CONVERSATION WITH IT. IN THIS CASE, WE HAVE A TASK WITHOUT ENTRIES. EXAMPLES WILL BE GIVEN LATER IN THE COURSE.

- BULLET #1 - THIS SHOWS THE TASK TYPE DECLARATION FOR A TASK TYPE WITHOUT ENTRIES.

- BULLET #2 - THIS SHOWS THE TASK DECLARATION FOR A ONE-OF-A-KIND TASK WITHOUT ENTRIES.

TASKS WITHOUT ENTRIES

- A TASK TYPE DECLARATION FOR TASKS WITHOUT ENTRIES:

  ```
  task type task type name is
  end task type name ;
  ```

  CAN BE ABBREVIATED AS:

  ```
  task type task type name ;
  ```

- A TASK DECLARATION FOR ONE-OF-A-KIND TASK OBJECTS WITHOUT ENTRIES

  ```
  task task object name is
  end task object name ;
  ```

  CAN BE ABBREVIATED AS:

  ```
  task task object name ;
  ```

- EXAMPLES:

  ```
  task type Alarm_Task_Type;
  task Control_Panel_Task;
  ```

INSTRUCTOR NOTES

- ALLOW 20 MINUTES FOR THE STUDENTS TO DO THE EXERCISE AND 10 MINUTES TO GO OVER IT.

EXERCISE 5.1 - TERMINAL HANDLER TASK TYPE

- WRITE THE TASK DECLARATION FOR A TERMINAL HANDLER TASK TYPE THAT HAS TWO USER DEFINED OPERATIONS:

  Get_Line   READS A LINE FROM THE TERMINAL AND RETURNS IT AS AN OUTPUT PARAMETER, ALONG WITH THE NUMBER OF CHARACTERS READ.

  Put_Line   TAKES A String INPUT AND WRITES IT TO THE TERMINAL.

- WRITE A SKELETON TASK BODY FOR THIS TYPE. THE BODY SHOULD CONTAIN AT LEAST THE FOLLOWING DECLARATIONS WHICH AN ALGORITHM MIGHT NEED:

  1. NAMED NUMBERS FOR THE MAXIMUM NUMBER OF LINES (24) PER SCREEN AND THE MAXIMUM NUMBER OF CHARACTERS (80) PER LINE.

  2. A TYPE DEFINITION FOR TYPE Buffer_Type.

  3. AN INPUT BUFFER DECLARATION AND AN OUTPUT BUFFER DECLARATION USING THE ABOVE TYPE.

  SHOW THE BRACKETING OF THE SEQUENCE OF STATEMENTS THAT A TERMINAL HANDLER TASK WILL EXECUTE. (THE STATEMENTS THEMSELVES CAN BE REPRESENTED AS "...")

- DECLARE A ONE-OF-A-KIND ARRAY OF TERMINAL HANDLERS. EACH COMPONENT OF THE ARRAY BELONGS TO THE TERMINAL HANDLER TASK TYPE. THE NUMBER OF TERMINALS IS GIVEN BY A CONSTANT (WHICH YOU MAY ASSUME IS ALREADY GIVEN) NAMED Max_Terminals.

INSTRUCTOR NOTES

- ALLOW 45 MINUTES FOR THIS SECTION.

- THIS SECTION GIVES THE RULES FOR TASK INITIATION AND TASK TERMINATION.

- THERE ARE TWO MAJOR POINTS IN THIS SECTION:

  1. TASKS BEGIN PARALLEL EXECUTION AT ABOUT THE TIME THEY ARE CREATED.

  2. DEPARTURE FROM A BLOCK, SUBPROGRAM, ETC. DEPENDS ON TERMINATION OF ANY TASKS DEPENDENT ON THE BLOCK, SUBPROGRAM, ETC.

- WE DO NOT INTRODUCE THE terminate ALTERNATIVE IN THIS SECTION NOR DO WE TALK ABOUT exceptions, SO OUR DEFINITIONS OF ACTIVATION, COMPLETION AND TERMINATION ARE SIMPLIFIED. THESE OTHER TOPICS WILL BE DISCUSSED IN LATER SECTIONS.

Section 6

TASK ACTIVATION AND TERMINATION

INSTRUCTOR NOTES

- BULLET 1 - THIS BULLET IS LARGELY MOTIVATIONAL. TELL THE CLASS THAT ACTIVATION AND TERMINATION IS THE Ada TERMINOLOGY.

- THE PURPOSE OF BULLETS 2 - 4 IS TWOFOLD:

  1. TO AVOID CONFUSION BETWEEN THE CYCLIC EXECUTIVE NOTION OF TASK INITIATION AND THE Ada NOTION OF TASK ACTIVATION.

  2. TO ASSURE THE STUDENT THAT Ada PROVIDES OTHER WAYS OF ACCOMPLISHING THE SAME THING:

     - WAYS TO ACHIEVE PERIODIC PROCESSING WITHOUT REPEATEDLY ACTIVATING TASKS;

     - WAYS TO DYNAMICALLY ACTIVATE TASKS, SHOULD THERE BE A LEGITIMATE NEED.

- BULLET 3 - THE VIEW DESCRIBED HERE IS USUALLY MORE APPROPRIATE BECAUSE IT ALLOWS EACH TASK TO PROCESS A SINGLE CONCEPTUAL THREAD.

- BULLET 4

  - ITEM 1: TIMED ENTRY CALLS AND DELAY ALTERNATIVES (DISCUSSED IN LATER SECTIONS) PROVIDE THIS ABILITY.

  - ITEM 2: DYNAMIC ACTIVATION BY ALLOCATING TASKS IS DISCUSSED IN THIS SECTION. DYNAMIC TERMINATION, BECAUSE OF A CALL ON A PARTICULAR ENTRY OR THROUGH AN ABORT STATEMENT, IS DISCUSSED LATER.

TASK ACTIVATION AND TERMINATION

- WHEN DEALING WITH TASKS WE NEED TO KNOW:
  - WHEN AND HOW ARE TASKS STARTED UP? - ACTIVATION
  - WHEN AND HOW DO TASKS FINISH? - TERMINATION

- TRADITIONAL VIEW (WITH CYCLIC EXECUTIVES):
  - A TASK IS "INITIATED" EACH TIME ITS TURN ARRIVES.
  - EACH INITIATION DOES THE WORK REQUIRED OF THE TASK ON ONE DUTY CYCLE, AND THEN TERMINATES.

- IN Ada PROGRAMS, ANOTHER VIEW IS *USUALLY MORE APPROPRIATE*:
  - A TASK FOR PERFORMING PERIODIC PROCESSING IS ACTIVATED ONCE, AT THE BEGINNING OF THE PROGRAM.
  - THE TASK EXECUTES A LOOP THAT IS REPEATED ONCE FOR EACH PROCESSING PERIOD.

- Ada PROVIDES WAYS TO
  - CONTROL THE INTERVALS AT WHICH THE PROCESSING LOOP IS REPEATED.
  - DYNAMICALLY ACTIVATE AND TERMINATE TASKS WHEN THAT ABILITY IS REQUIRED FOR SOME OTHER REASON.

INSTRUCTOR NOTES

- JUST BEFORE THE SEQUENCE OF STATEMENTS OF Dispatch_Output ARE EXECUTED, THE Operator_Console TASK AND THE TWO Remote_Terminal_Handler TASKS ARE ACTIVATED. (ACTUALLY THIS OCCURS JUST AFTER THE DECLARATIONS ARE ELABORATED. TRY TO STAY AWAY FROM TALKING ABOUT ELABORATION FOR NOW.)

- AFTER Dispatch_Output FINISHES ITS STATEMENTS, IT MUST STILL WAIT UNTIL THE THREE TASKS TERMINATE.

- TELL THE CLASS THAT THE SAME WOULD HOLD IF WE HAD USED A BLOCK OR OUTER TASK INSTEAD OF A PROCEDURE.

DECLARED TASKS

```
with Terminal_Handler_Package;

procedure Dispatch_Output is

     Number_Of_Remote_Terminals : constant := 2;
          ...

     Operator_Console        : Terminal_Handler_Package.Terminal_Handler_Type;
     Remote_Terminal_Handler : array (1 .. Number_Of_Remote_Terminals) of
                                  Terminal_Handler_Package.Terminal_Handler_Type;

begin

     -- Operator_Console, Remote_Terminal_Handler (1) AND
     --   Remote_Terminal_Handler (2) START HERE.

          ...

     -- Dispatch_Output WAITS FOR Operator_Console,
     --    Remote_Terminal_Handler (1), AND
     --    Remote_Terminal_Handler (2) TO TERMINATE

end Dispatch_Output;
```

INSTRUCTOR NOTES

- THIS SLIDE GIVES THE RULES FOR ACTIVATION AND TERMINATION OF TASKS DECLARED WITHIN A SUBPROGRAM, BLOCK OR OUTER TASK. WE WILL SHOW LATER THAT WE WANT TO DISTINGUISH BETWEEN COMPLETION AND TERMINATION.

- MENTION TO THE CLASS THAT IF A RECORD OBJECT HAS A TASK OBJECT AS A COMPONENT, THEN ALLOCATION OF THE TASK OBJECT PROCEEDS AS IF THE TASK WAS AN ENTIRE VARIABLE.

DECLARED TASKS - CONTINUED

- WHEN A TASK IS DECLARED IN THE DECLARATIVE PART OF A SUBPROGRAM, BLOCK, OR OUTER TASK:
  - IT IS ACTIVATED JUST BEFORE THE CORRESPONDING SEQUENCE OF STATEMENTS.
  - THE SUBPROGRAM, BLOCK, OR OUTER TASK CANNOT BE LEFT UNTIL THE TASK TERMINATES.

- LEAVING A SUBPROGRAM INCLUDES
  - EXECUTING A RETURN STATEMENT
  - COMPLETING AN EXCEPTION HANDLER
  - PROPAGATING AN EXCEPTION

- LEAVING A BLOCK INCLUDES
  - EXECUTING A RETURN STATEMENT
  - EXECUTING AN EXIT OR GOTO STATEMENT OUT OF THE BLOCK
  - COMPLETING AN EXCEPTION HANDLER
  - PROPAGATING AN EXCEPTION

INSTRUCTOR NOTES

- THIS IS THE FIRST OF TWO SLIDES ON ALLOCATED TASKS. THIS IS A SIMPLE EXAMPLE. A MORE COMPLEX EXAMPLE FOLLOWS.

- Operator_Console_Pointer.all IS ACTIVATED WHEN IT IS ALLOCATED.

- AFTER Dispatch_Output.all FINISHES ITS EXECUTION, IT MUST STILL WAIT FOR Operator_Console_Pointer.all TO TERMINATE.

- TELL THE CLASS THAT IF Operator_Console_Pointer HAD BEEN GIVEN AN INITIAL VALUE, IT WOULD BE ALLOCATED AS IF IT WAS A DECLARED OBJECT.

ALLOCATED TASKS -- EXAMPLE 1

```
with Terminal_Handler_Package;

procedure Dispatch_Output is

       ...

   type Terminal_Handler_Pointer_Type is
       access Terminal_Handler_Package.Terminal_Handler_Type;

   Operator_Console_Pointer : Terminal_Handler_Pointer_Type;

       ...

begin

       ...

   Operator_Console_Pointer :=
                            new Terminal_Handler_Package.Terminal_Handler_Type;

                     -- Operator_Console_Pointer.all STARTS HERE

       ...

                     -- Dispatch_Output WAITS FOR Operator_Console_Pointer.all
                     --    TO TERMINATE
end Dispatch_Output;
```

INSTRUCTOR NOTES

- THIS EXAMPLE OF ALLOCATED TASKS SHOWS THAT A SUBPROGRAM, BLOCK OR OUTER TASK CONTAINING AN ACCESS TYPE DESIGNATING SOME TASK TYPE, MUST WAIT UNTIL ALL TASK OBJECTS DESIGNATED BY THE TASK TYPE TERMINATE.

- NOTE THAT WE ARE ONLY USING THE <u>USE</u> CLAUSE TO MAKE THINGS FIT ON THE SLIDE.

- ALL OF THE TASKS ARE ACTIVATED WHEN THEY ARE ALLOCATED.

- WHEN THE BLOCK FINISHES, IT MUST STILL WAIT FOR Message_Buffer_Pointer.all TO TERMINATE. THE BLOCK DOES NOᵀ NEED TO WAIT FOR Remote_Terminal_Handler_Pointer.all TO TERMINATE, SINCE ITS ACCESS TYPE WAS NOT DEFINED IN THE BLOCK.

ALLOCATED TASKS - CONTINUED

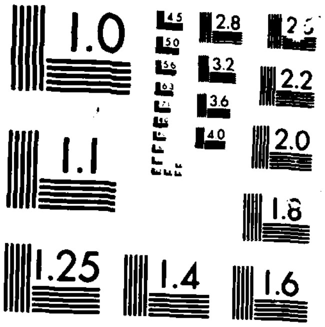
```
with Terminal_Handler_Package; use Terminal_Handler_Package;
with Message_Buffer_Package; use Message_Buffer_Package;
procedure Dispatch_Output is
       ...
   type Terminal_Handler_Pointer_Type is access Terminal_Handler_Type;
   Operator_Console_Pointer : Terminal_Handler_Pointer_Type;
       ...
begin
       ...
   Operator_Console_Pointer := new Terminal_Handler_Type;
                     -- Operator_Console_Pointer.all STARTS HERE
       ...
   declare
          ...
      type Message_Buffer_Pointer_Type is access Message_Buffer_Type;
      Message_Buffer_Pointer          : Message_Buffer_Pointer_Type;
      Remote_Terminal_Handler_Pointer : Terminal_Handler_Pointer_Type;
   begin
          ...
      Message_Buffer_Pointer := new Message_Buffer_Type;
                     -- Message_Buffer_Pointer.all STARTS HERE
          ...
      Remote_Terminal_Handler_Pointer := new Terminal_Handler_Type;
                     -- Remote_Terminal_Handler_Pointer.all STARTS HERE
          ...
                     -- BLOCK WAITS FOR Message_Buffer_Pointer.all TO TERMINATE
   end; -- block
          ...
                     -- Dispatch_Output WAITS FOR Operator_Console_Pointer.all
                     --    AND Remote_Terminal_Handler.all TO TERMINATE
end Dispatch_Output;
```

INSTRUCTOR NOTES

- TELL THE CLASS THAT IF A RECORD OBJECT HAS A TASK OBJECT AS A COMPONENT, THEN ALLOCATION PROCEEDS AS IF THE TASK WERE DESIGNATED BY AN ACCESS VALUE.

AD-A166 351 ADA (TRADE NAME) TRAINING CURRICULUM REAL-TIME SYSTEMS 3/5
IN ADA L401 TEACHER'S GUIDE VOLUME 1(U) SOFTECH INC
WALTHAM MA 1986 DAAB07-83-C-K514

UNCLASSIFIED F/G 5/9 NL



MICROCOPY RESOLUTION TEST CHART

ALLOCATED TASKS - CONTINUED

- A DYNAMICALLY ALLOCATED TASK IS ACTIVATED WHEN THE ALLOCATOR FOR THE TASK IS EVALUATED.

- A SUBPROGRAM, BLOCK OR OUTER TASK CONTAINING AN ACCESS TYPE DEFINITION CANNOT BE LEFT UNTIL ALL ALLOCATED TASKS DESIGNATED BY THIS TYPE HAVE TERMINATED.

  - A SUBPROGRAM, BLOCK OR OUTER TASK THAT DYNAMICALLY ALLOCATES A TASK NEED NOT WAIT FOR THE TASK TO TERMINATE, UNLESS IT ALSO CONTAINS THE ACCESS TYPE DEFINITION.

INSTRUCTOR NOTES

- WHEN THE DECLARATIVE PART OF Dispatch_Output IS ELABORATED, SO IS Remote_Package. THE ELABORATION OF THE PACKAGE INCLUDES THE ELABORATION OF THE DECLARATIVE PART OF THE PACKAGE BODY AND THE EXECUTION OF ITS SEQUENCE OF INITIALIZATION STATEMENTS.

- Remote_Message_Buffer, DECLARED IN THE PACKAGE SPECIFICATION, AND Remote_Terminal_Handler, DECLARED WITHIN THE PACKAGE BODY, ARE ACTIVATED - AT THE SAME TIME - AFTER ELABORATION OF THE PACKAGE BODY'S DECLARATIVE PART, BUT BEFORE EXECUTION OF ITS SEQUENCE OF INITIALIZATION STATEMENTS.

- ONCE THE DECLARATIVE PART OF Dispatch_Output HAS BEEN ELABORATED, Operator_Consule IS ACTIVATED - THE TASKS FROM THE PACKAGE ARE ALREADY ACTIVE.

- FOR PURPOSES OF TERMINATION, THE TASKS ARE TREATED AS IF THEY HAD BEEN DECLARED OUTSIDE OF THE PACKAGE.

TASKS AND (NON-LIBRARY) PACKAGES

```
with Terminal_Handler_Package, Message_Buffer_Package;
procedure Dispatch_Output is
       ...
   Operator_Console : Terminal_Handler_Package.Terminal_Handler_Type;
   package Remote_Package is
       ...
     Remote_Message_Buffer : Message_Buffer_Package.Message_Buffer_Type;
   end Remote_Package;

   package body Remote_Package is
       ...
     Remote_Terminal_Handler : Terminal_Handler_Package.Terminal_Handler_Type;
       ...
   begin
                     -- Remote_Terminal_Handler AND Remote_Message_Buffer
                     --    START HERE.
          ...
   end Remote_Package;

       ...
begin
                     -- Operator_Console STARTS HERE.
       ...
                     -- Dispatch_Output WAITS FOR Operator_Console,
                     --    Remote_Terminal_Handler AND Remote_Message_Buffer
                     --    TO TERMINATE.
end Dispatch_Output;
```

INSTRUCTOR NOTES

- LIBRARY PACKAGES ARE TREATED DIFFERENTLY AND WILL BE PRESENTED AT THE END OF THIS SECTION.

- POINT OUT THAT FOR TERMINATION PURPOSES, THE PACKAGE IS TREATED AS IF IT IS INVISIBLE.

TASKS AND (NON-LIBRARY) PACKAGES - CONTINUED

- WHEN A TASK IS DECLARED WITHIN A PACKAGE SPECIFICATION OR THE DECLARATIVE PART OF A PACKAGE BODY

  - THE TASK IS ACTIVATED JUST AFTER ALL DECLARATIONS IN THE PACKAGE SPECIFICATION AND PACKAGE BODY HAVE BEEN ELABORATED. (IF THERE IS A SEQUENCE OF INITIALIZATION STATEMENTS, THIS MEANS JUST BEFORE THESE STATEMENTS ARE EXECUTED.)

  - THE PROGRAM UNIT CONTAINING THE PACKAGE MUST WAIT FOR THE TASK TO TERMINATE.

- WHEN A TASK IS DYNAMICALLY ALLOCATED AND ITS ACCESS TYPE APPEARS WITHIN A PACKAGE SPECIFICATION OR PACKAGE DECLARATION THEN

  - IT IS ACTIVATED WHEN THE ALLOCATOR FOR THE TASK IS EVALUATED.

  - THE PROGRAM UNIT CONTAINING THE PACKAGE CANNOT BE LEFT UNTIL THE ALLOCATED TASK TERMINATES.

INSTRUCTOR NOTES

- IN ORDER TO FULLY EXPLAIN TERMINATION, WE NEED TO TALK ABOUT MASTERS AND DEPENDENCY.

- WHEN, SAY, A PROGRAM IS THE MASTER OF A TASK, THIS MEANS THAT THE SUBPROGRAM MUST WAIT FOR THE TASK TO TERMINATE BEFORE IT CAN BE LEFT. IN PARTICULAR, A MASTER CAN FINISH ITS OWN EXECUTION BUT STILL HAVE TO WAIT FOR ONE OR MORE OF ITS DEPENDENT TASKS TO TERMINATE. ON THE NEXT SLIDE WE DESCRIBE WHAT IT MEANS FOR A MASTER TO FINISH.

- WHEN A TASK DEPENDS ON A MASTER, THIS MEANS THAT THERE IS A "POTENTIAL" FOR THE TASK TO DEPEND ON DECLARATIONS WITHIN THE MASTER. THE TASK MIGHT NOT DEPEND ON THESE DECLARATIONS, BUT THE POTENTIAL IS THERE. THE MASTER CANNOT "GO AWAY" UNTIL THIS POTENTIAL DEPENDENCY CEASES.

MASTERS AND DEPENDENCY

- EACH TASK DEPENDS ON A MASTER, WHICH MAY BE
  - A TASK
  - A CURRENTLY EXECUTING BLOCK OR SUBPROGRAM
  - A LIBRARY PACKAGE (A PACKAGE DECLARED WITHIN ANOTHER PROGRAM UNIT IS NEVER A MASTER)
- A DECLARED TASK DEPENDS ON THE MASTER WHOSE EXECUTION CREATES THE TASK.
- A DYNAMICALLY ALLOCATED TASK DEPENDS ON THE MASTER CONTAINING THE CORRESPONDING ACCESS TYPE DEFINITION.
- CONTROL DOES NOT LEAVE A MASTER UNTIL ALL TASKS DEPENDING ON THE MASTER HAVE TERMINATED.

INSTRUCTOR NOTES

- THIS SLIDE DESCRIBES WHEN A MASTER FINISHES EXECUTING.

COMPLETION

- A SUBPROGRAM HAS REACHED COMPLETION OR COMPLETED WHEN EITHER
  - IT HAS FINISHED EXECUTING ITS CORRESPONDING SEQUENCE OF STATEMENTS, OR
  - IT REACHES A RETURN STATEMENT
  - IT RAISES AN UNHANDLED EXCEPTION
  - IT FINISHES EXECUTION OF AN EXCEPTION HANDLER

- A BLOCK STATEMENT HAS REACHED COMPLETION OR COMPLETED WHEN EITHER
  - IT HAS FINISHED EXECUTING ITS CORRESPONDING SEQUENCE OF STATEMENTS, OR
  - IT REACHES
    - A RETURN STATEMENT, OR
    - AN EXIT OR GOTO STATEMENT OUT OF THE BLOCK.
  - IT RAISES AN UNHANDLED EXCEPTION
  - IT FINISHES EXECUTION OF AN EXCEPTION HANDLER

- A TASK HAS REACHED COMPLETION OR COMPLETED WHEN
  - IT HAS FINISHED EXECUTING ITS CORRESPONDING SEQUENCE OF STATEMENTS
  - IT RAISES AN UNHANDLED EXCEPTION
  - IT FINISHES EXECUTION OF AN EXCEPTION HANDLER

INSTRUCTOR NOTES

- FOR THE INSTRUCTOR ONLY: WE DO NOT INTRODUCE THE TERMINATE ALTERNATIVE UNTIL LATER IN THE COURSE, AT THAT WE WILL NEED TO EXPAND ON THIS DEFINITION.

TERMINATION

- A TASK REACHES TERMINATION OR HAS TERMINATED IF EITHER:
  - IT HAS NO DEPENDENT TASKS AND HAS COMPLETED, OR
  - IT HAS DEPENDENT TASKS AND
    - IT HAS COMPLETED, AND
    - ALL OF ITS DEPENDENT TASKS HAVE TERMINATED.
- A BLOCK OR SUBPROGRAM THAT HAS COMPLETED IS NOT LEFT UNTIL ALL OF ITS DEPENDENT TASKS TERMINATE.

INSTRUCTOR NOTES

- THIS IS A QUICK IN CLASS QUIZ.

- FOR EACH DECLARED OR ALLOCATED TASK ASK THE CLASS TO:

  1. NAME ITS MASTER,
  2. TELL WHEN THE TASK IS ACTIVATED.

- Message_Buffer HAS Test_Procedure AS ITS MASTER AND IS ACTIVATED JUST BEFORE THE SEQUENCE OF STATEMENTS IN Test_Procedure IS EXECUTED.

- Shared_Count_One_Pointer.all AND Shared_Count_Two_Pointer.all EACH HAVE Test_Procedure AS A MASTER, AND EACH IS ACTIVATED WHEN IT IS ALLOCATED.

- Alternate_Message_Buffer HAS THE BLOCK AS ITS MASTER AND IS ACTIVATED JUST BEFORE THE SEQUENCE OF STATEMENTS OF THE BLOCK IS EXECUTED.

- THE BLOCK WAITS ONLY FOR Alternate_Message_Buffer.

- AS THE CLASS PROVIDES THE ANSWERS, MAKE SURE YOU WRITE THEM ON THE SLIDE!

FILL IN THE BLANKS

```
procedure Test_Procedure is

   Message_Buffer : Message_Buffer_Type;
                                                              -- MASTER IS __________
   type Shared_Count_Pointer_Type is access Shared_Count_Type;
   Shared_Count_One_Pointer : Shared_Count_Pointer_Type;

begin -- Test_Procedure
      ...

   declare
       Shared_Count_Two_Pointer : Shared_Count_Pointer_Type;
       Alternate_Message_Buffer : Message_Buffer_Type;
                                                              -- MASTER IS __________

   begin
         ...

       Shared_Count_One_Pointer := new Shared_Count_Type;
                                                              -- MASTER IS __________
       Shared_Count_Two_Pointer := new Shared_Count_Type;
                                                              -- MASTER IS __________
         ...
            -- WAIT FOR ______________________________ TO TERMINATE

   end; -- block

         ...
            -- WAIT FOR ______________________________ TO TERMINATE

end Test_Procedure;
```

INSTRUCTOR NOTES

- ABOVE THE LINE IS A LIBRARY PACKAGE, WHILE BELOW THE LINE IS A MAIN PROCEDURE THAT USES IT.

- A LIBRARY PACKAGE (UNLIKE NON-LIBRARY PACKAGES) IS A MASTER, SO THE MASTER OF Message_Buffer IS Library_Package.

- Library_Packages ARE ELABORATED BEFORE THE MAIN PROGRAM IS EXECUTED, SO Message_Buffer IS ACTIVATED BEFORE Main_Procedure IS EXECUTED.

- Message_Buffer_Pointer.all HAS Library_Package AS ITS MASTER, SINCE THAT IS THE MASTER CONTAINING THE ACCESS TYPE DEFINITIONS.

- Main_Procedure IS ONLY THE MASTER OF Shared_Count, THEREFORE Main_Procedure ONLY NEEDS TO WAIT FOR Shared_Count TO TERMINATE.

- THE LANGUAGE SPECIFICATION DOES NOT EVEN SAY IF Message_Buffer OR Message_Buffer_Pointer.all HAVE TO TERMINATE.

MAIN PROGRAMS AND LIBRARY UNITS

```
package Library_Package is -- A LIBRARY PACKAGE
       task type Shared_Count_Type is ...
       task type Message_Buffer_Type is ...
       type Message_Buffer_Pointer_Type is access Message_Buffer_Type;
       ...
end Library_Package;

package body Library_Package is
       ...
   Message_Buffer : Message_Buffer_Type;  -- MASTER IS Library_Package
       ...
begin
              -- Message_Buffer STARTS BEFORE Main_Procedure IS EXECUTED.
       ...
end Library_Package;
```

```
with Library_Package;

procedure Main_Procedure is
       ...
   Shared_Count           : Library_Package.Shared_Count_Type;
                                                 -- MASTER IS Main_Procedure
   Message_Buffer_Pointer : Library_Package.Message_Buffer_Pointer_Type;
                                                 -- MASTER IS Library_Package
       ...
begin
              -- Shared_Count STARTS HERE
   Message_Buffer_Pointer := new Library_Package.Message_Buffer_Type;
              -- Message_Buffer_Pointer.all STARTS HERE
       ...
              -- Main_Procedure WAITS FOR Shared_Count TO TERMINATE, BUT
              -- NOT FOR Message_Buffer OR Message_Buffer_Pointer.all
end Main_Procedure;
```

INSTRUCTOR NOTES

- THIS SLIDE GIVES RULES FOR LIBRARY PACKAGES.
- MAKE SURE THAT THE CLASS UNDERSTANDS THE DIFFERENCE BETWEEN THE RULES FOR TASKS DECLARED IN LIBRARY PACKAGES AND TASKS DECLARED IN NON-LIBRARY PACKAGES.

ACTIVATION - LIBRARY PACKAGES AND MAIN PROGRAMS

- LIBRARY PACKAGES ARE ELABORATED BEFORE THE MAIN PROGRAM IS EXECUTED.
- FOR TASKS DECLARED WITHIN A PACKAGE SPECIFICATION, OR THE DECLARATIVE PART OF A PACKAGE BODY
  - ACTIVATION OCCURS JUST AFTER ALL THE DECLARATIONS IN THE PACKAGE SPECIFICATION AND PACKAGE BODY HAVE BEEN ELABORATED. (IF THERE IS A SEQUENCE OF INITIALIZATION STATEMENTS THIS MEANS JUST BEFORE THE SEQUENCE OF STATEMENTS ARE EXECUTED.)
  - THIS OCCURS BEFORE THE MAIN PROGRAM IS EXECUTED.
- THERE IS NO MASTER RELATIONSHIP BETWEEN LIBRARY PACKAGES AND THE MAIN PROGRAM.
  - THE MAIN PROGRAM DOES <u>NOT</u> WAIT FOR TERMINATION OF
    - TASKS DECLARED IN LIBRARY UNITS
    - ALLOCATED TASKS WHOSE ACCESS TYPE IS DECLARED IN A LIBRARY PACKAGE.

INSTRUCTOR NOTES

PART III

TASK COOPERATION

7. SIMPLE RENDEZVOUS

8. SELECTIVE WAITS

9. SELECT STATEMENTS FOR MAKING ENTRY CALLS

10. AVOIDING DEADLOCK

INSTRUCTOR NOTES

- ALLOW 45 MINUTES FOR THIS SECTION.

- THIS SECTION

  1. SUMMARIZES THE FORM OF AN ENTRY DECLARATION AND THE FORM OF AN ENTRY CALL (THESE WERE USED INFORMALLY IN THE PREVIOUS SECTIONS);

  2. INTRODUCES THE ACCEPT STATEMENT;

  3. INTRODUCES THE RENDEZVOUS CONCEPT.

- THE MAIN MESSAGE OF THIS SECTION IS THAT TASKS COMMUNICATE WHEN AN ACCEPT STATEMENT ACCEPTS AN ENTRY CALL.

- RECOMMEND THAT STUDENTS READ EXERCISE 3.1 IN THE REAL-TIME ADA WORKBOOK.

Section 7

SIMPLE RENDEZVOUS

INSTRUCTOR NOTES

- IN SECTION 6, WE USED ENTRY CALLS AS IF THEY WERE PROCEDURE CALLS. IN THIS SECTION, WE START TO DESCRIBE THEM VIA THE RENDEZVOUS MECHANISM.

- THIS SLIDE SUMMARIZES EVERYTHING THE STUDENT NEEDS TO KNOW ABOUT THE FORM OF AN ENTRY DECLARATION OR CALL - EXCEPT FOR ENTRY FAMILIES.

- DO NOT SPEND TOO MUCH TIME WITH THIS SLIDE, UNLESS SOME STUDENTS ARE HAVING TROUBLE.

- MENTION THAT .all IS NOT NEEDED IN THE CALL TO THE Receive ENTRY OF Message_Buffer_Pointer.

REVIEW OF ENTRY DECLARATIONS AND CALLS

- ENTRY DECLARATIONS

  - SYNTAX:

    ```
    entry  entry name  [( formal parameters )]
    ```

  - EXAMPLES:

    ```
    task type Message_Buffer_Type is
       entry Send (Message : in Message_Type);
       entry Receive (Message : out Message_Type);
    end Message_Buffer_Type;
    ```

- ENTRY CALLS

  - SYNTAX:

    ```
    task object name . entry name [( actual parameters )];

    access value . entry name [( actual parameters )];
    ```

    THE access value MUST POINT TO A TASK OBJECT.

    THE TASK OBJECT NAME OR ACCESS VALUE MAY BE

    - AN IDENTIFIER
    - AN EXPANDED NAME
    - A COMPONENT OF AN ARRAY OR RECORD
    - A FUNCTION CALL
    - AN ALLOCATED OBJECT (.all)
    - ETC.

  - EXAMPLES:

    ```
    Message_Buffer.Send (Message => Short_Message);

    Message_Buffer_Pointer.all.Receive (Message => Long_Message);

        -- Equivalent to Message_Buffer_Pointer.Receive (Long_Message)
    ```

INSTRUCTOR NOTES

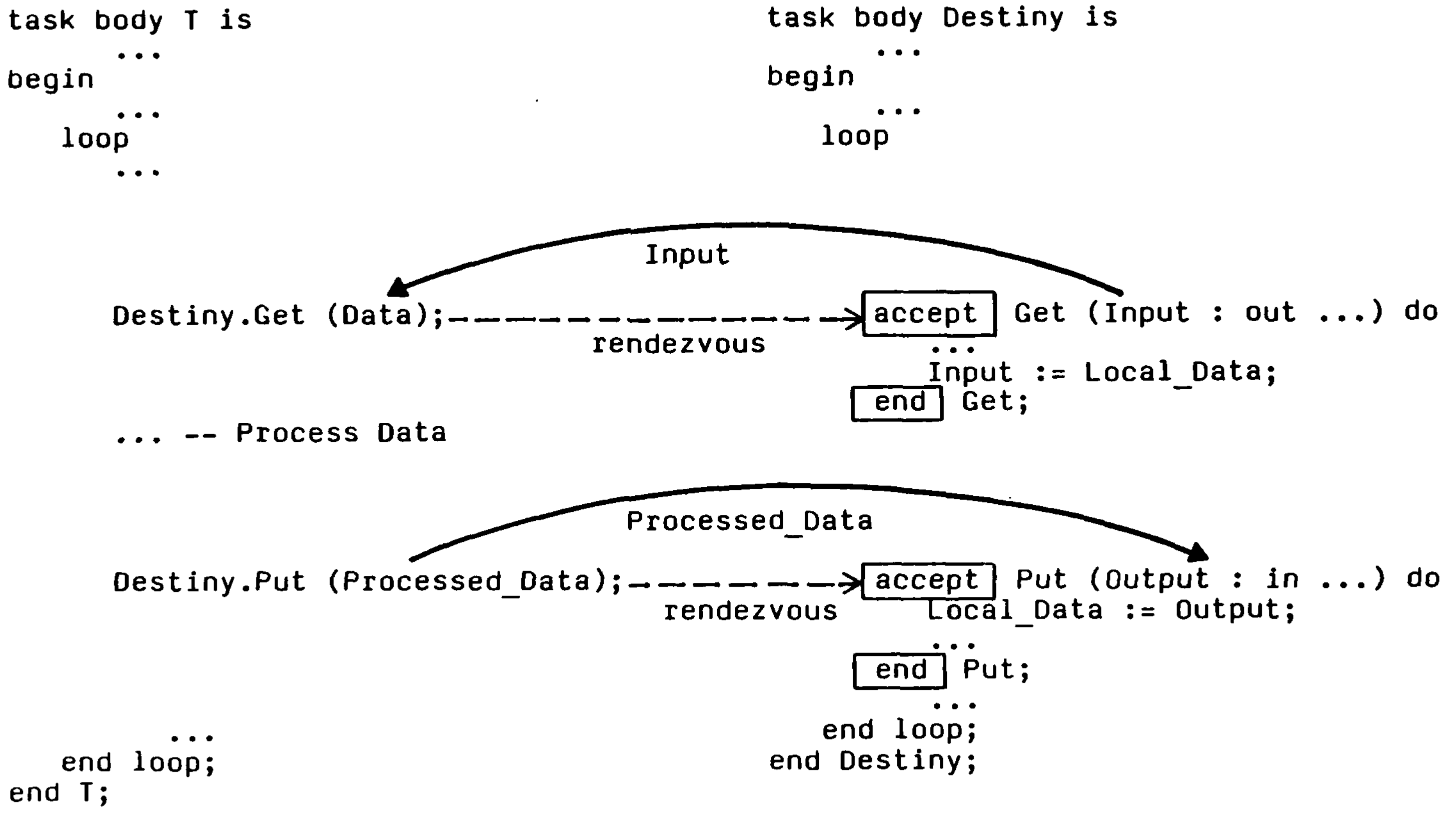
- THIS SLIDE INTRODUCES THE ACCEPT STATEMENT AND THE CONCEPT OF RENDEZVOUS. TELL THE STUDENTS THAT MORE DETAILS WILL BE PROVIDED LATER. THE ARROWS SHOW THE RENDEZVOUS AND THE FLOW OF INFORMATION.
- TAKE THE STUDENTS THROUGH THE FOLLOWING STEPS:

  1. WHEN TASK T WANTS TO GET DATA FROM TASK Destiny IT ISSUES THE Get ENTRY CALL. WHEN TASK Destiny IS READY TO GIVE DATA IT EXECUTES THE ACCEPT STATEMENT.
     - SINCE BOTH TASKS ARE READY TO COMMUNICATE WITH EACH OTHER, THEY ENTER INTO A RENDEZVOUS.
     - TASK T WAITS WHILE TASK Destiny EXECUTES ITS ACCEPT STATEMENT FOR ENTRY Get.
     - THE STATEMENTS WITHIN THE ACCEPT STATEMENT ARE EXECUTED; THE FINAL ONE ASSIGNS THE VALUE OF Local_Data TO THE FORMAL PARAMETER Input.
     - WHEN THE ACCEPT STATEMENT HAS FINISHED EXECUTING, THE VALUE OF THE FORMAL PARAMETER Input IS COPIED TO THE ACTUAL PARAMETER Data.
     - THE RENDEZVOUS IS COMPLETE, THE TASKS HAVE COMPLETED THEIR COMMUNICATION, AND EACH PROCEEDS ASYNCHRONOUSLY.

     WHEN TASK T WANTS TO SEND DATA TO TASK Destiny IT ISSUES THE Put ENTRY CALL. WHEN TASK Destiny IS READY TO Receive THE DATA IT EXECUTES THE ACCEPT STATEMENT.
     - SINCE BOTH TASKS ARE READY TO COMMUNICATE WITH EACH OTHER, THEY ENTER INTO A RENDEZVOUS.
     - TASK T WAITS WHILE TASK Destiny EXECUTES ITS ACCEPT STATEMENT FOR ENTRY Put.
     - THE CONTENTS OF THE ACTUAL PARAMETER Processed_Data IS COPIED INTO THE FORMAL PARAMETER Output.
     - THE STATEMENTS WITHIN THE ACCEPT STATEMENT ARE EXECUTED, WITH THE FIRST ONE ASSIGNING THE VALUE OF THE FORMAL PARAMETER Output TO THE LOCAL VARIABLE Local_Data.
     - THE RENDEZVOUS IS COMPLETE, THE TASKS HAVE COMPLETED THEIR COMMUNICATION, AND EACH PROCEEDS ASYNCHRONOUSLY.

A RENDEZVOUS WITH DESTINY

- WHEN TWO TASKS WANT TO COMMUNICATE, ONE ISSUES AN ENTRY CALL AND THE OTHER ACCEPTS IT.
- WHEN THE TASKS ACTUALLY COMMUNICATE, THEY ARE SAID TO BE IN RENDEZVOUS.
- EXAMPLE:



```
task body T is                                  task body Destiny is
         ...                                             ...
begin                                           begin
         ...                                             ...
   loop                                            loop
         ...

                                        Input
        Destiny.Get (Data);--------------------->accept Get (Input : out ...) do
                                 rendezvous            ...
                                                       Input := Local_Data;
                                                      end Get;
        ... -- Process Data

                                     Processed_Data
        Destiny.Put (Processed_Data);---------->accept Put (Output : in ...) do
                                    rendezvous         Local_Data := Output;
                                                       ...
                                                      end Put;
                                                       ...
          ...                                      end loop;
   end loop;                                    end Destiny;
end T;
```

INSTRUCTOR NOTES

- NOW TELL THE CLASS THAT WHEN THE PREVIOUS TASKS WANT TO COMMUNICATE, EITHER ONE MAY BE READY FIRST.

- THIS SLIDE ILLUSTRATES THE THREE CASES THAT CAN OCCUR IN THIS SIMPLE RENDEZVOUS. FOR NOW, IT IS ENOUGH TO TELL THE CLASS THAT WHICHEVER TASK IS READY FIRST JUST WAITS UNTIL THE OTHER IS READY. WE WILL EXPAND ON THIS IN A LATER SLIDE.

- NOTE THAT WE HAVE RENAMED Destiny AS D.

WAITING FOR RENDEZVOUS

```
 ________________________________________   T ready but must
                                            wait for D to accept
 Ⓣ Get (Data)
    -------->                         Ⓓ
     call

 Ⓣ Get (Data)           accept
    -------->         --------> Ⓓ
     call             Get (Data)

          Get (Data)
 Ⓣ ======================        Ⓓ
       rendezvous
 ________________________________________   D ready to accept
                                            but must wait for T
                        accept
 Ⓣ                    --------> Ⓓ
                      Get (Data)

 Ⓣ Get (Data)           accept
    -------->         --------> Ⓓ
     call             Get (Data)

          Get (Data)
 Ⓣ ======================        Ⓓ
       rendezvous
 ________________________________________   T and D ready
 Ⓣ                               Ⓓ          simultaneously
                                            so neither waits
 Ⓣ Get (Data)           accept
    -------->         --------> Ⓓ
     call             Get (Data)

          Get (Data)
 Ⓣ ======================        Ⓓ
       rendezvous
```

INSTRUCTOR NOTES

- ALTHOUGH THE entry name AT THE END OF THE ACCEPT STATEMENT IS OPTIONAL, GOOD PROGRAMMING STYLE SUGGESTS THAT IT SHOULD APPEAR.

- EXAMPLE 3 ILLUSTRATES THAT UNLIKE PROCEDURES, MORE THAN ONE ACCEPT STATEMENT MAY EXIST FOR AN ENTRY DECLARATION. EACH ACCEPT STATEMENT HANDLES A DIFFERENT CALL TO THE ENTRY.

- FOR THE INSTRUCTORS' BENEFIT ONLY IF A STUDENT SHOULD ASK. IN THE ABBREVIATED FORM OF THE ACCEPT STATEMENT THERE IS NO SEQUENCE OF STATEMENTS BUT FORMAL PARAMETERS ARE STILL ALLOWED. WHILE THIS MIGHT SEEM STRANGE, IT IS POSSIBLE THAT A TASK MAY WANT TO ACCEPT AN ENTRY AT ONE POINT AND USE THE PARAMETERS, WHILE AT ANOTHER POINT, THE TASK MAY SIMPLY WANT TO ACCEPT THE ENTRY CALL BUT IGNORE THE PARAMETERS. IF SUCH AN ACCEPT STATEMENT APPEARS WITHIN A TASK, IT IS POSSIBLE THAT THIS IS AN ERROR, I.E., THAT THE PROGRAMMER JUST FORGOT TO WRITE IN THE SEQUENCE OF STATEMENTS. AN IMPLEMENTOR COULD CHOOSE TO HAVE THE COMPILER WARN YOU ABOUT SUCH AN ACCEPT STATEMENT.

ACCEPT STATEMENT

SYNTAX:

```
     accept entry name [(formal parameters)] do
              sequence of statements
     end [entry name];
```

EXAMPLES

```
     1.     accept Increase_Count (By: in Positive; New_Sum: out Positive) do
               Sum := Sum + By;
               New_Sum := Sum;
            end Increase_Count;

     2.     accept Request do
               Resource_Available := False;
            end Request;

     3.     accept Deliver (Item : in Item_Type) do
               First_Item := Item;
            end Deliver;
               ...
            loop
               accept Deliver (Item : in Item_Type) do
                   Next_Item := Item;
               end Deliver;
               ...
            end loop;
```

AN ACCEPT STATEMENT OF THE FORM:

```
     accept entry name [(formal parameters)] do
        null;
     end [entry name];
```

CAN BE ABBREVIATED AS: accept entry name [(formal parameters)];

INSTRUCTOR NOTES

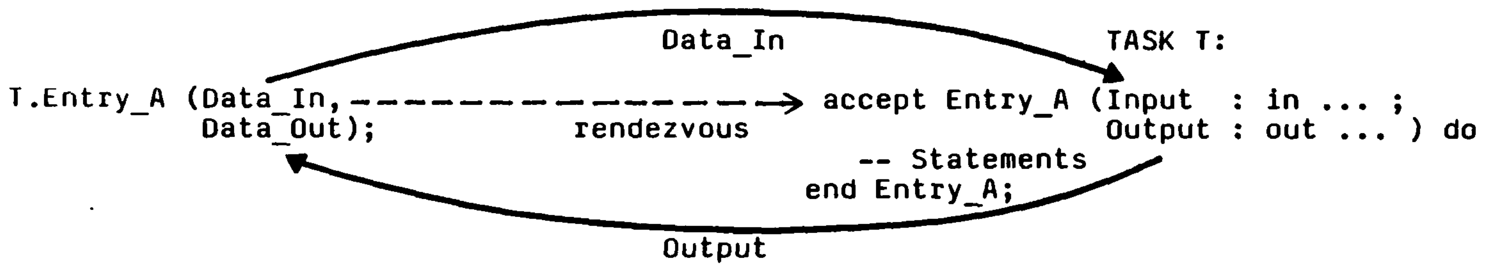
- THIS SLIDE DEFINES A RENDEZVOUS MORE PRECISELY.
- THE NEXT TWO SLIDES GIVE EXAMPLES AND PICTORIAL EXPLANATIONS ABOUT MORE INVOLVED RENDEZVOUS.
- TELL THE CLASS THAT THERE ARE SEVERAL OPTIMIZATIONS THAT AN IMPLEMENTATION MAY MAKE IN ORDER TO MAKE RENDEZVOUS MORE EFFICIENT. TELL THE CLASS THAT WE WILL COVER SOME OF THEM IN THE LAST PART OF THE CLASS.
- WHEN GOING THROUGH THIS SLIDE, REFER TO THE EXAMPLE AT THE BOTTOM OF THE SLIDE. FOR EXAMPLE,
    - WHEN EXPLAINING THE SECOND ITEM IN THE FIRST BULLET, REFER TO THE CALL TO Entry_A AND SAY THAT THIS CALL PUTS THE CALLING TASK ON A QUEUE FOR Entry_A.
    - WHEN EXPLAINING STEP 1 IN A RENDEZVOUS POINT OUT THAT THE ACTUAL PARAMETER Data_In IS COPIED TO THE FORMAL in PARAMETER Input.

RENDEZVOUS

- AN ENTRY CALL IS EXECUTED WHEN THE CALLING TASK AND THE CALLED TASK HAVE BOTH ARRIVED AT A POINT WHERE THEY EXPECT COMMUNICATION TO TAKE PLACE. THIS IS CALLED A RENDEZVOUS.
  - EACH ENTRY HAS A QUEUE ASSOCIATED WITH IT.
  - WHEN A TASK MAKES AN ENTRY CALL, IT GOES ON THE QUEUE FOR THAT ENTRY.
  - WHEN AN ACCEPT STATEMENT IS EXECUTED BY A TASK, THE TASK IT RENDEZVOUS WITH IS REMOVED FROM THE QUEUE OF THE ENTRY THAT WAS ACCEPTED.
  - IF THE CALLED TASK REACHES AN ACCEPT STATEMENT FIRST, IT WAITS FOR SOME TASK TO CALL THAT ENTRY.
  - IF A TASK ISSUES AN ENTRY CALL BEFORE THE CALLED TASK REACHES AN ACCEPT STATEMENT, THE CALLING TASK WAITS.
  - SEVERAL DIFFERENT TASKS MIGHT CALL THE SAME ENTRY OF THE SAME TASK BEFORE THE CALLED TASK REACHES AN ACCEPT STATEMENT FOR THAT ENTRY. EACH CALLING TASK WAITS, AND ENTRY CALLS ARE ACCEPTED IN ORDER OF ARRIVAL.

- STEPS IN A RENDEZVOUS:
  1. THE in AND in out ACTUAL PARAMETERS OF THE ENTRY CALL ARE COPIED INTO THE FORMAL PARAMETERS OF THE ACCEPT STATEMENT.
  2. THE STATEMENTS INSIDE ACCEPT STATEMENT ARE EXECUTED.
  3. THE in out AND out FORMAL PARAMETERS OF THE ACCEPT STATEMENT ARE COPIED BACK TO THE ACTUAL PARAMETERS OF THE ENTRY CALL, COMPLETING EXECUTION OF THE ENTRY CALL STATEMENT.

- FOLLOWING A RENDEZVOUS, BOTH TASKS RESUME ASYNCHRONOUS EXECUTION.



```
                                                TASK T:
T.Entry_A (Data_In,----------------> accept Entry_A (Input  : in ... ;
         Data_Out);      rendezvous                 Output : out ... ) do
                                          -- Statements
                                       end Entry_A;
```

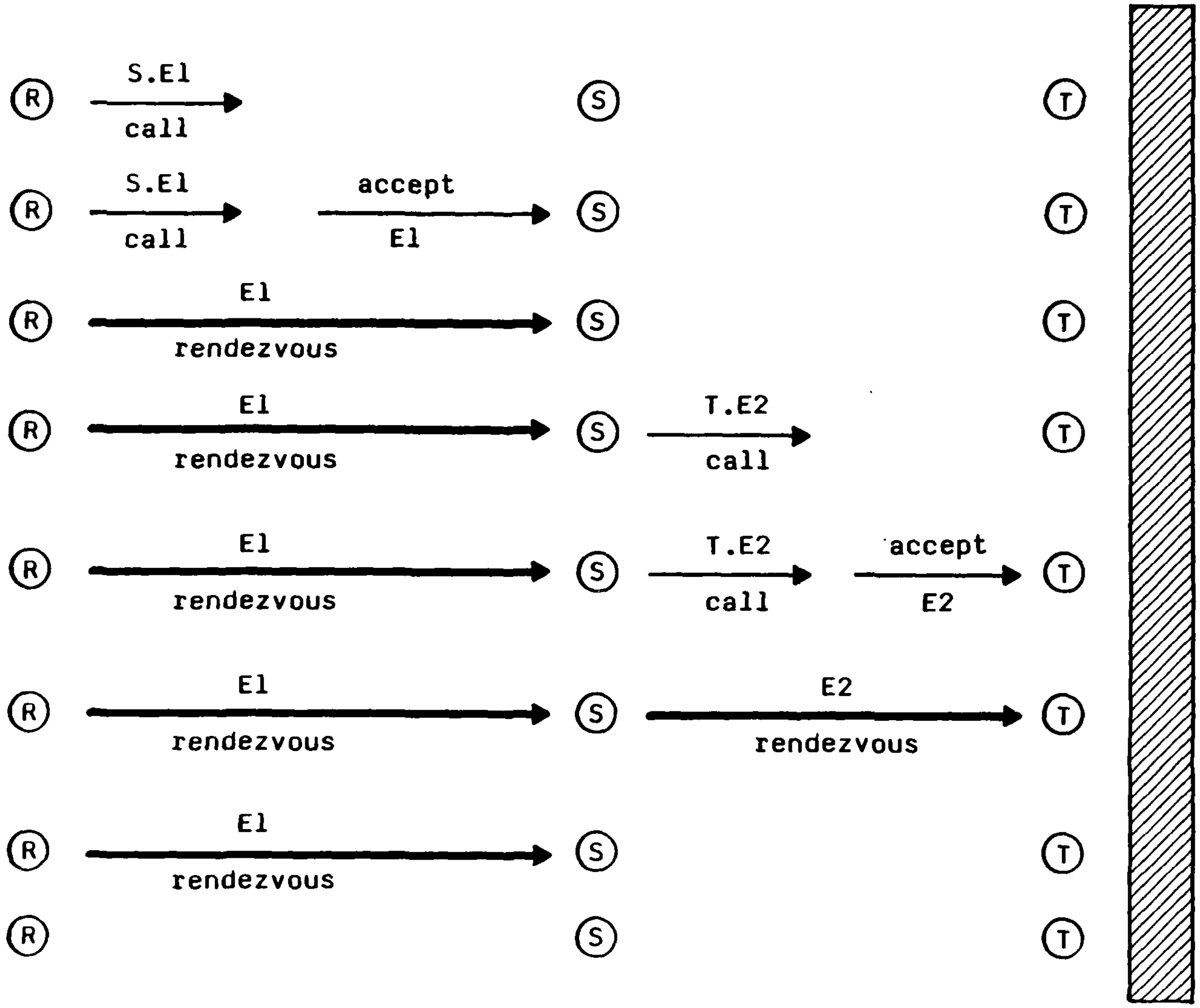
INSTRUCTOR NOTES

- THIS SLIDE ILLUSTRATES THAT A CALLED TASK CAN ESTABLISH ADDITIONAL RENDEZVOUS EVEN IF IT IS ALREADY IN THE MIDDLE OF A RENDEZVOUS. THE SITUATION SHOWN IS THE CALLING TASK MAKING AN ENTRY CALL TO A THIRD TASK.

- THE FIGURES ON THIS SLIDE ILLUSTRATE WHAT IS HAPPENING IN THE CODE SKELETON ON THE RIGHT HAND SIDE OF THE SLIDE. THE NEXT PARAGRAPH EXPLAINS THIS IN DETAIL.

- TASK S ACCEPTS AN ENTRY CALL ENTRY E1 FROM Task R, WHICH RESULTS IN THE RENDEZVOUS OF Task R AND Task S. IN ORDER TO SATISFY Task R, Task S MAKES AN ENTRY CALL ENTRY E2 TO Task T, IN THE MIDDLE OF ITS RENDEZVOUS WITH Task R. WHEN Task T ACCEPTS THIS CALL, Task S AND Task T ARE IN RENDEZVOUS. AT THIS POINT, Task S IS IN RENDEZVOUS WITH TWO DIFFERENT TASKS. IT HAS ACCEPTED ENTRY E1 AND HAS MADE THE ENTRY CALL ENTRY E2. Task S MUST COMPLETE ITS RENDEZVOUS WITH Task T BEFORE IT CAN COMPLETE ITS RENDEZVOUS WITH Task R.

RENDEZVOUS WITHIN RENDEZVOUS



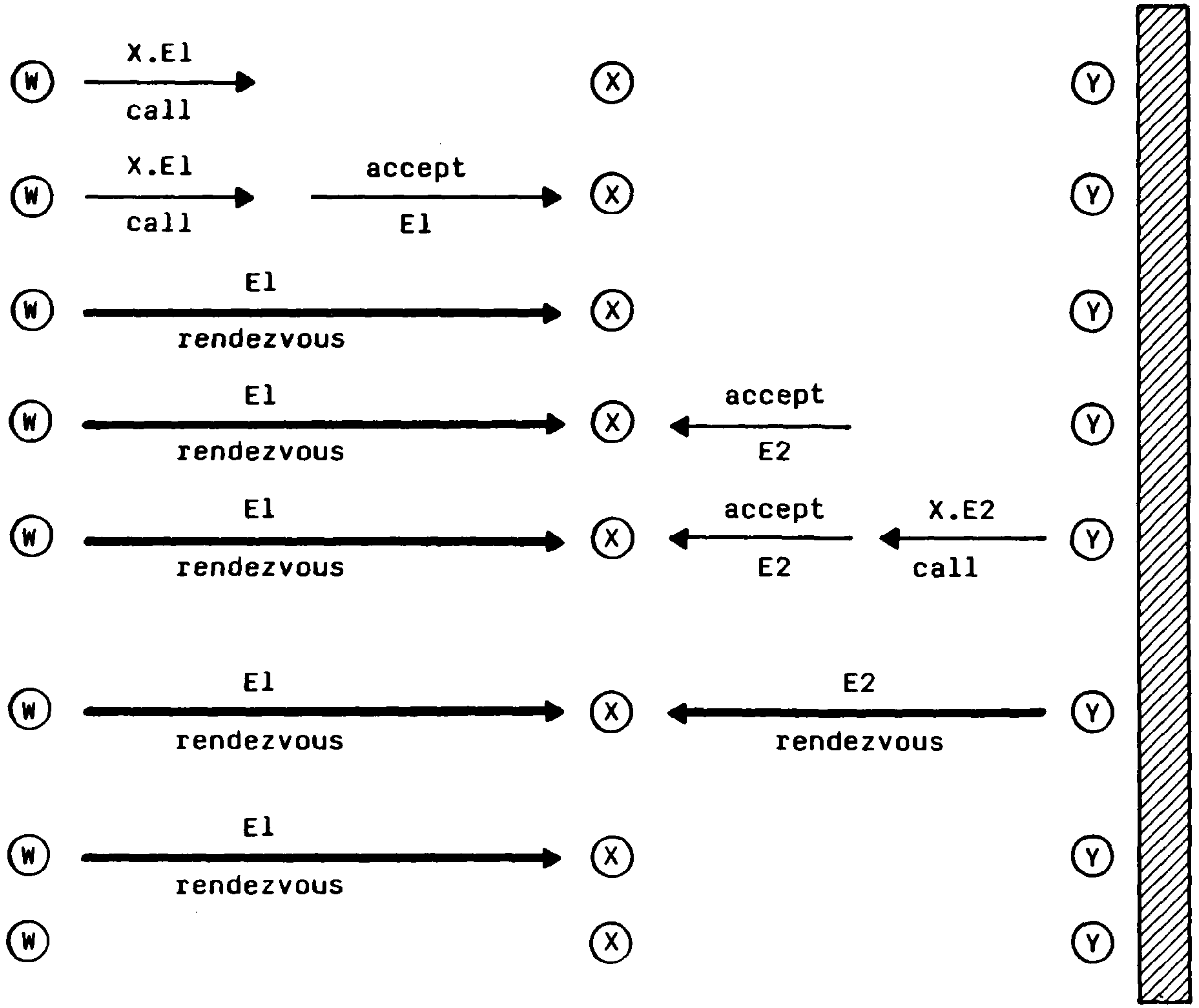
```
task body S is
   ...
begin
   ...
   accept E1 do
      ...
      T.E2;
   end E1;
   ...
end S;
```

INSTRUCTOR NOTES

- THIS SLIDE ILLUSTRATES ANOTHER EXAMPLE OF A CALLING TASK TAKING PART IN A SECOND RENDEZVOUS WHILE IN THE MIDDLE OF ANOTHER ONE. THIS TIME, THE CALLING TASK ACCEPTS AN ENTRY CALL FROM A THIRD TASK. THIS SHOWS THAT THE SEQUENCE OF STATEMENTS IN AN ACCEPT STATEMENT MAY INCLUDE OTHER ACCEPT STATEMENTS.

- THIS SLIDE SHOWS A SITUATION THAT IS A LITTLE DIFFERENT FROM THE PREVIOUS SLIDE. HERE, Task W MAKES AN ENTRY CALL ENTRY E1 TO Task X, AND Task X ACCEPTS THIS ENTRY, SO THAT Task W AND Task X ARE IN RENDEZVOUS. IN THE MIDDLE OF THIS RENDEZVOUS, Task X ACCEPTS AN ENTRY CALL ENTRY E2 FROM Task Y. IT MUST COMPLETE THESE RENDEZVOUS IN THE REVERSE ORDER IN WHICH THEY WERE ESTABLISHED.

RENDEZVOUS WITHIN RENDEZVOUS - DEJA VU



```
task body X is
   ...
begin
   ...
   accept El do
      ...
      accept E2 do

         ...
      end E2;
   end El;
end X;
```

INSTRUCTOR NOTES

- THIS IS INTRODUCED AS A SIMPLE CASE OF A COMPLETE TASK TYPE.
- THIS SLIDE REVISITS SIMULTANEOUS UPDATE, SO BRIEFLY REVIEW THE PROBLEM.
- THE LOCK SOLUTION IS GOOD AS AN EXAMPLE BUT THE MONITOR SOLUTION IN SECTION 12 IS BETTER.
- MAKE SURE THE STUDENTS UNDERSTAND HOW THIS EXAMPLE SOLVES THE SIMULTANEOUS UPDATE PROBLEM.

SIMULTANEOUS UPDATE USING LOCKS - A PRIMITIVE SOLUTION

- THE PROBLEM OF SIMULTANEOUS UPDATE WAS DESCRIBED IN SECTION 3. A PRIMITIVE SOLUTION USES LOCKS.

```
task body Lock_Type is
      begin
         loop
            accept Lock;
              -- Another task calling  Lock at this
              --   point will have to wait until
              --   Unlock has been called so this
              --   task can again accept Lock.
            accept Unlock;
         end loop;
      end Lock_Type;
```

- EXAMPLE:

```
Database_Lock : Lock_Type;
       ...
Database_Lock.Lock;
  -- The calling task now has exclusive use of the
  --   database.  Other tasks calling Database_Lock.Lock
  --   must wait.
Database_Lock.Unlock;
  -- Now other calls on Database_Lock.Lock can be
  --   accepted.
```

- A MORE ABSTRACT AND LESS ERROR PRONE SOLUTION USES <u>MONITORS</u>, WHICH WILL BE INTRODUCED LATER.

INSTRUCTOR NOTES

- THIS IS A SHORT EXERCISE TO SEE IF THE CLASS UNDERSTANDS THE MATERIAL THAT HAS BEEN COVERED SO FAR.

- THE SOLUTION IS A STRAIGHTFORWARD APPLICATION OF THE Lock_Type. GIVE THE CLASS ABOUT 10 MINUTES TO SOLVE THE PROBLEM AND ALLOW ABOUT 10 MINUTES TO DISCUSS IT.

```
task Track_Manager is
   entry Obtain_Track;
   entry Release_Track;
end Track_Manager;
task body Track_Manager is
begin
   loop
       accept Obtain_Track;
                            -- at this point, the train can proceed safely
                            --    across the track.
       accept Release_Track;
                            -- now another train can get access to the track.
   end loop;
end Track_Manager;

task type Train_Type is
end Train_Type;
task body Train_Type is
       ...
begin
       ...
   Track_Manager.Obtain_Track;
   Cross_River;
   Track_Manager.Release_Track;
       ...
end Train_Type;
```

EXERCISE 7.1 - THE TRAIN PROBLEM REVISITED

- SOUTHERN PATHETIC RAILROAD HAS HIRED YOU TO MODERNIZE ITS TRACK SHARING SCHEME.

  - THE FOLLOWING REQUIREMENTS STILL HOLD:

    - THERE SHOULD BE NO POSSIBILITY OF ONCOMING TRAINS COLLIDING.
    - THE SCHEME SHOULD NOT DEADLOCK.
    - EASTBOUND AND WESTBOUND TRAINS SHOULD BE TREATED EQUALLY IN THE LONG RUN.

  HOWEVER, SOUTHERN PATHETIC WANTS TO CHANGE THE WAY ENGINEERS COMMUNICATE. THE CURRENT SCHEME HAS INCREASED MEDICAL EXPENSES DUE TO EYESTRAIN AMONG ENGINEERS (FROM LOOKING FOR FLAGS GOING UP AND DOWN).

    - THE RAILROAD WANTS A THIRD INDEPENDENT AGENT TO MANAGE THE TRACK.
    - WHEN A TRAIN IS READY TO CROSS THE TRACK, THE ENGINEER NOTIFIES THE TRACK MANAGER. THE ENGINEER WAITS UNTIL THE TRACK MANAGER SIGNALS THAT THE TRAIN MAY CROSS.

- WRITE AN Ada PROGRAM TO SPECIFY WHAT THE TRACK MANAGER AND EACH ENGINEER SHOULD DO.

- USE ONLY THE TASKING FEATURES INTRODUCED IN THIS SECTION.

- YOU MAY ALSO CALL ONE OR MORE OF THE FOLLOWING SUBPROGRAMS:

```
procedure Raise_Flag;                       function Boulder_Present
procedure Lower_Flag;                          return Boolean;
function Oncoming_Flag_Raised               procedure Pick_Up_Boulder;
   return Boolean;                          procedure Drop_Boulder;

procedure Cross_River;      -- Throw the switches and use the shared track.
procedure Wait;             -- Do nothing for some fixed amount of time.
```

INSTRUCTOR NOTES

- BULLET #2

  - ITEM 1 - ACTUALLY, NESTED ACCEPT STATEMENTS HAVE ONE RESTRICTION. AN ACCEPT STATEMENT FOR, SAY ENTRY_E, MAY NOT CONTAIN ANOTHER ACCEPT STATEMENT FOR THE SAME ENTRY_E. DO NOT BRING THIS UP IN CLASS UNLESS EXPLICITLY ASKED.

  - ITEM 2 - TELL THE STUDENTS THAT IT ALSO CANNOT APPEAR WITHIN A PACKAGE OR TASK NESTED WITHIN THE TASK BODY.

PLACEMENT OF ACCEPT STATEMENTS

- AN ACCEPT STATEMENT FOR AN ENTRY OF A TASK MAY ONLY APPEAR WITHIN THE TASK BODY OF THE TASK.

- WITHIN THE TASK BODY, AN ACCEPT STATEMENT FOR AN ENTRY OF THE TASK:

  - MAY APPEAR WITHIN ANY COMPOUND STATEMENT, E. G., BLOCK STATEMENT, LOOP STATEMENT, IF STATEMENT, ACCEPT STATEMENT, ETC.

  - MAY NOT APPEAR WITHIN A SUBPROGRAM NESTED WITHIN THE TASK BODY.

INSTRUCTOR NOTES

- THIS SLIDE ILLUSTRATES THE ILLEGAL USE OF AN accept STATEMENT INSIDE A SUBPROGRAM BODY.

  THE PROBLEM IS AS FOLLOWS: Ground_Speed_Task IS TO REPEATEDLY ACCEPT CALLS ON Deliver_Doppler_Ground_Speed TO OBTAIN GROUND SPEED READINGS FOR AN AIRCRAFT, AVERAGE SEVERAL CONSECUTIVE READINGS TOGETHER, AND CALL A PROCEDURE Display_Ground_Speed TO DISPLAY THE AVERAGED READING. THE NUMBER OF VALUES IN THE AVERAGE DEPENDS ON THE ALTITUDE. AT LOW ALTITUDES, INDIVIDUAL READINGS ARE MORE RELIABLE AND IT IS MORE CRUCIAL TO HAVE CURRENT DATA ON THE DISPLAY, SO FEWER READINGS ARE USED TO COMPUTE AN AVERAGE.

  IT IS TEMPTING TO WRITE A FUNCTION THAT COMPUTES THE AVERAGE OF n CONSECUTIVE READINGS, WHERE n IS GIVEN BY A PARAMETER. INDEED, THIS WOULD NORMALLY BE CONSIDERED GOOD PROGRAMMING STYLE. THIS IS THE APPROACH TAKEN ON THE SLIDE, AND IT MAKES THE if STATEMENT VERY EASY TO FOLLOW. UNFORTUNATELY THE Average_Reading FUNCTION BODY IS ILLEGAL.

  THE NEXT SLIDE SHOWS A WAY TO WRITE THE TASK BODY LEGALLY, WITHOUT ENCLOSING AN ACCEPT STATEMENT IN A SUBPROGRAM.

PLACEMENT OF ACCEPT STATEMENTS - AN INCORRECT EXAMPLE

SINCE THE ACCEPT STATEMENT FOR Deliver_Doppler_Ground_Speed IS WITHIN A FUNCTION NESTED IN THE TASK, THE FOLLOWING EXAMPLE IS INCORRECT.

```
task body Ground_Speed_Task is
             ...

      function Average_Reading (Number_Of_Readings : in Positive) return Speed_Type is
         Sum : Speed_Type := 0.0;
      begin -- Average_Reading
         for I in 1 .. Number_Of_Readings loop
             accept Deliver_Doppler_Ground_Speed (Speed : in Speed_Type) do
                Sum := Sum + Speed;                      -- Illegal accept statement
             end Deliver_Doppler_Ground_Speed;           --   inside a subprogram body

         end loop;
         return Sum/Speed_Type (Number_Of_Readings);
      end Average_Reading;

begin -- Ground_Speed_Task

      loop
         if Altitude in 0.0 .. 50.0 then
             Ground_Speed := Average_Reading (Number_Of_Readings => 1);
         elsif Altitude in 50.0 .. 100.0 then
             Ground_Speed := Average_Reading (Number_Of_Readings => 2);
         elsif Altitude in 100.0 .. 200.0 then
             Ground_Speed := Average_Reading (Number_Of_Readings => 3);
         elsif Altitude in 200.0 .. 500.0 then
             Ground_Speed := Average_Reading (Number_Of_Readings => 5);
         else
             Ground_Speed := Average_Reading (Number_Of_Readings => 10);
         end if;
         Display_Ground_Speed (Ground_Speed);
      end loop;

end Ground_Speed_Task;
```

INSTRUCTOR NOTES

- THIS SLIDE CORRECTS THE PREVIOUS ONE. DO NOT SPEND MUCH TIME ON IT. THE POINT IS SIMPLY THAT IT IS POSSIBLE TO "PROGRAM AROUND" THE NEED FOR A SUBPROGRAM CALLED FROM WITHIN A TASK BODY.

- TELL THE STUDENTS THAT PROBLEMS SUCH AS THIS CAN OFTEN BE SOLVED BY REVERSING THE DIRECTION OF THE RENDEZVOUS. THIS IS DISCUSSED IN SECTION 11.

PLACEMENT OF ACCEPT STATEMENTS - A CORRECT EXAMPLE

```
loop
   if Altitude in 0.0 .. 50.0 then
      Number_Of_Readings := 1;
   elsif Altitude in 50.0 .. 100.0 then
      Number_Of_Readings := 2;
   elsif Altitude in 100.0 .. 200.0 then
      Number_Of_Readings := 3;
   elsif Altitude in 200.0 .. 500.0 then
      Number_Of_Readings := 5;
   else
      Number_Of_Readings := 10;
   end if;

   Ground_Speed := 0.0;

   for I in 1 .. Number_Of_Readings loop
      accept Deliver_Doppler_Ground_Speed (Speed : in Speed_Type) do
         Ground_Speed := Ground_Speed + Speed;
      end Deliver_Doppler_Ground_Speed;
   end loop;
   Ground_Speed := Ground_Speed / Speed_Type (Number_Of_Readings);
   Display_Ground_Speed (Ground_Speed);
end loop;
```

INSTRUCTOR NOTES

- ALLOW 120 MINUTES FOR PRESENTING THE MATERIAL IN THIS SECTION.

- ALLOW AN ADDITIONAL 100 MINUTES FOR 5 IN CLASS EXERCISES THAT APPEAR THROUGHOUT THIS SECTION.

- THIS SECTION DESCRIBES THE SELECTIVE WAIT STATEMENT.

- THE MAIN MESSAGES IN THIS SECTION ARE:

  1. TASKS CAN BE WRITTEN *NONDETERMINISTICALLY*.
  2. TASKS CAN EXERCISE VARYING DEGREES OF CONTROL OVER THE ENTRY CALLS THEY ACCEPT AND WHEN THEY ACCEPT THEM.
  3. TASKS CAN LIMIT OR AVOID WAITS FOR A RENDEZVOUS.

- FOR EACH FORM OF THE SELECTIVE WAIT STATEMENT, THE PROBLEM TO BE SOLVED WILL FIRST BE INTRODUCED, THEN THE FORM OF THE SELECTIVE WAIT, AND THEN THE SOLUTION.

- THE TYPE Duration WILL BE ALSO BE INTRODUCED.

Section 8

SELECTIVE WAITS

INSTRUCTOR NOTES

- THIS SLIDE JUST INFORMS THE STUDENTS THAT THERE ARE SEVERAL WAYS THAT THE entry CALL / accept MODEL WE SAW EARLIER CAN BE EXTENDED. THIS SECTION ONLY DISCUSSES THE ONES FROM THE CALLED SIDE, I.E. ONLY FOR THE ACCEPTING TASK.

- HOPEFULLY, THIS WILL KEEP STUDENTS FROM JUMPING AHEAD AND ASKING QUESTIONS ABOUT THE CALLING SIDE.

- BULLET #2 - FOR THE INSTRUCTOR ONLY

  - ITEM 1 - THIS IS A SIMPLE SELECTIVE WAIT; NOTE: ARBITRARY SELECTION DOES NOT IMPLY FAIRNESS (MORE ON FAIRNESS LATER).

  - ITEM 2 - THIS IS A SELECTIVE WAIT WITH A GUARDED ALTERNATIVE

  - ITEM 3 - THIS IS A SELECTIVE WAIT WITH A DELAY ALTERNATIVE

  - ITEM 4 - THIS IS A TIMED ENTRY CALL

  - ITEM 5 - THIS IS A CONDITIONAL ENTRY CALL

select STATEMENTS - OVERVIEW

- THE LIMITED FORM OF THE ENTRY CALL / accept STATEMENT MODEL WE HAVE SEEN SO FAR IS ADEQUATE FOR A TASK WHOSE ENTRY CALLS ARE STRICTLY ORDERED - I.E., DETERMINISTIC TASKS.

- THIS MODEL IS INADEQUATE WHEN, FOR EXAMPLE:

  - A TASK MAY BE CAPABLE OF ACCEPTING ANY ONE OF SEVERAL ENTRY CALLS AT SOME INSTANT - IT IS ARBITRARY WHICH ONE IS SELECTED FIRST.
  - A TASK MAY BE CAPABLE OF ACCEPTING ANY ONE OF SEVERAL ENTRY CALLS AT SOME INSTANT, BUT MAY NEED TO GUARD AGAINST CERTAIN ENTRIES BEING ACCEPTED IF SOME CONDITION DOES NOT HOLD.
  - IF A TASK'S ENTRIES ARE NOT CALLED WITHIN SOME SPECIFIED TIME, THE TASK MAY NEED TO TAKE CORRECTIVE ACTION OR RAISE AN ALARM.

  TASK IS A CALLED TASK

  - IF A TASK'S ENTRY CALL IS NOT ACCEPTED WITHIN SOME SPECIFIED TIME, THEN THE TASK MAY WISH TO PERFORM SOME OTHER ACTION.
  - IF A TASK'S ENTRY CALL CANNOT BE ACCEPTED IMMEDIATELY, THEN THE TASK MAY WISH TO PERFORM SOME OTHER ACTION.

  TASK IS A CALLING TASK

SUCH TASKS ARE CALLED NONDETERMINISTIC.

- Ada PROVIDES FOR THESE CASES AND MORE WITH select STATEMENTS OF THE FORM:

  selective wait

  conditional entry call

  timed entry call

- WE WILL LOOK AT selective wait STATEMENTS IN THIS SECTION.

INSTRUCTOR NOTES

- THIS SLIDE DESCRIBES WHY STRICT ORDERING OF ENTRY CALLS CAN HINDER CONCURRENCY OR EVEN PROHIBIT THE PROBLEM FROM BEING SOLVED.

- THIS IS MOTIVATION FOR THE SIMPLE SELECTIVE WAIT.

- MANY DETAILS FOR THE SELECTIVE WAIT WILL BE POSTPONED UNTIL THE END OF THIS SECTION.

PROBLEM 1 - SELECTING ONE OF SEVERAL ENTRY CALLS

- ASSUME Task_C HAS TWO ENTRIES - Entry_A AND Entry_B - AND THAT Task_C ACCEPTS ENTRY CALLS IN THE ORDER Entry_A, Entry_B, Entry_A, Entry_B, ... Task_C MIGHT CONTAIN A CODE FRAGMENT SUCH AS:

```
     ...
loop
   accept Entry_A (...) do
     ...
   end Entry_A;
     ...     -- STATEMENTS AFTER Entry_A
   accept Entry_B (...) do
     ...
   end Entry_B;
     ...     -- STATEMENTS AFTER Entry_B
end loop;
```

- TASK_C ILLUSTRATES TWO POTENTIAL PROBLEMS.

  - CONCURRENCY MAY BE HINDERED:

    - ASSUME Entry_A AND Entry_B ARE CALLED BY Task_A AND Task_B, RESPECTIVELY.
    - TASK_B MUST WAIT FOR Task_A TO COMPLETE ITS RENDEZVOUS WITH Entry_A BEFORE Task_B CAN RENDEZVOUS WITH Entry_B, AND SIMILARLY FOR Task_A. IF THE PROBLEM BEING SOLVED DOES NOT REQUIRE ALTERNATION, THEN THIS JUST SLOWS DOWN Task_A AND Task_B.

  - THE PROBLEM MAY NOT ALLOW ALTERNATION:

    - ASSUME Task_C IS CALLED BY A SINGLE TASK S.
    - TASK S MUST HAVE THE CALLING SEQUENCE Entry_A, Entry_B, Entry_A, Entry_B, ETC. THIS EXCLUDES TASKS WHOSE CALLS ON THE ENTRIES OF Task_C ARE NOT PREDICTABLE.

INSTRUCTOR NOTES

- THIS SLIDE SHOWS HOW THE PREVIOUS PROBLEM CAN BE SOLVED MORE REASONABLY.

- LAST BULLET - WE ARE ASSUMING THAT THE TIME SPENT WITHIN A PASS THROUGH THE LOOP IN Task_C IS NEGLIGIBLE WITH RESPECT TO Task_B'S SPEED.

PROBLEM 1 - REVISITED

- WE MODIFY Task_C TO REMOVE THE STRICT ORDERING IN THE CALLING OF Task_C'S ENTRIES.
- Task_C MAY NOW CONTAIN A CODE FRAGMENT SUCH AS:

```
          ...
loop
   select
       accept Entry_A (...) do
       ...
       end Entry_A;
       ...    -- STATEMENTS AFTER Entry_A
   or
       accept Entry_B (...) do
       ...
       end Entry_B;
       ...    -- STATEMENTS AFTER Entry_B
   end select;
end loop;
```

- IF Entry_A and Entry_B ARE CALLED BY Task_A AND Task_B, RESPECTIVELY, THEN NEITHER Task_B NOR Task_A SLOWS DOWN THE OTHER.
- Task_C MAY NOW BE CALLED BY A SINGLE TASK WHOSE CALLS ON Entry_A and Entry_B ARE NOT PREDICTABLE.

INSTRUCTOR NOTES

- WE NOW GIVE THE FULL DEFINITION OF THE Shared_Count_Type DISCUSSED EARLIER IN THE COURSE.
- NOTICE THE RENAMING TO ADD CLARITY TO THE Increase_Count ACCEPT STATEMENT.

AN EXAMPLE - Shared_Count_Type

```
task type Shared_Count_Type is
   entry Increase_Count (By : in Positive);
   entry Get_Count (Sum_So_Far : out Natural);
end Shared_Count_Type;

task body Shared_Count_Type is
   Sum : Natural := 0;
begin -- Shared_Count_Type

   loop
      select
         accept Increase_Count (By : in Positive) do

             declare
                Increment : Positive renames By;
             begin
                Sum := Sum + Increment;
             end; -- block

         end Increase_Count;

      or

         accept Get_Count (Sum_So_Far : out Natural) do
             Sum_So_Far := Sum;
         end Get_Count;

      end select;
   end loop;
end Shared_Count_Type;
```

INSTRUCTOR NOTES

- THIS SLIDE GIVES A SIMPLE INTRODUCTION TO THE SELECTIVE WAIT STATEMENT.

- NOTE THAT ONLY ACCEPT ALTERNATIVES ARE INTRODUCED.

SIMPLE SELECTIVE WAIT

- SIMPLE SELECTIVE WAIT STATEMENTS ALLOW TASKS TO ACCEPT CALLS ON ANY ONE OF SEVERAL SPECIFIED ENTRIES. FORM OF THE STATEMENT IS:

```
          select
               accept alternative
          {or
               accept alternative }
          end select;
```

WHERE accept alternative HAS THE FORM:

```
      accept statement
    [ sequence of statements ]
```

- IN EXECUTING THE SELECTIVE WAIT, ONE OF TWO CASES CAN OCCUR:

1. NONE OF THE ENTRIES HAS BEEN CALLED:
   - WAIT UNTIL ONE IS CALLED AND ACCEPT THAT CALL.
2. ONE OR MORE OF THE ENTRIES HAS BEEN CALLED:
   - ONE ACCEPT STATEMENT THAT CAN BE EXECUTED IMMEDIATELY IS SELECTED ARBITRARILY AND EXECUTED.

THE METHOD OF ARBITRARY SELECTION DEPENDS ON THE RUNTIME SYSTEM, BUT A GOOD RUNTIME SYSTEM WILL BE FAIR. THE PROGRAMMER SHOULD VIEW THE CHOICE AS RANDOM.

INSTRUCTOR NOTES

- THE FOLLOWING LOOP MIGHT BE PART OF AN ONBOARD NAVIGATION SYSTEM.
- A PILOT SETS A DESTINATION WHICH RESULTS IN THE Set_Destination ENTRY BEING CALLED.
- PERIODICALLY, THE CURRENT POSITION IS REPORTED, AT WHICH TIME IT IS COMPARED WITH THE CURRENT DESTINATION. IF THE CURRENT POSITION IS WITHIN RANGE OF THE DESTINATION, THE PILOT IS ALERTED.

EXAMPLE OF A SIMPLE SELECTIVE WAIT

```
-- Assume Current_Position and Current_Destination have initial values at this point.

loop

   select

      accept Set_Destination (Destination : in Position_Type) do
         Current_Destination := Destination;
      end Set_Destination;

   or

      accept Report (New_Position : in Position_Type) do
         Current_Position := New_Position;
      end Report;

      if Within_Range (Current_Position, Current_Destination) then
         Alert_Pilot;
      end if;

   end select;

end loop;
```

INSTRUCTOR NOTES

- THIS IS THE FIRST OF FIVE EXERCISES WITHIN THIS SECTION. ALLOW 20 MINUTES - 15 MINUTES TO DO THE EXERCISE, AND 5 MINUTES TO GO OVER IT.

- THIS EXERCISE IS INTENDED TO DISCOVER WHETHER THOSE STUDENTS THAT WERE NODDING DURING THE PRESENTATION REALLY UNDERSTAND THE MATERIAL.

- MAKE SURE THE CLASS IS TOLD THAT WE WILL RETURN TO THIS EXAMPLE FOUR MORE TIMES, SO THEY SHOULD LEAVE LOTS OF SPACE FOR CHANGES.

EXERCISE 8.1 - SIMPLE SELECTIVE WAIT

- WRITE A TASK THAT ALLOWS SEVERAL TASKS TO CHANGE AND USE AN INTEGER COUNTER.

- THE FOLLOWING ENTRIES MUST BE PROVIDED:

  1. SET THE COUNTER'S INITIAL VALUE

  2. INCREMENT COUNTER BY ONE

  3. DECREMENT COUNTER BY ONE

  4. REPORT CURRENT COUNTER VALUE

INSTRUCTOR NOTES

- THIS SLIDE ILLUSTRATES THE NEED FOR GUARDS.

- THE ALGORITHM ON THE SLIDE IS AN ATTEMPT TO SHARE A POOL OF BUFFERS. AS DESCRIBED IN A LATER BULLET, IT FAILS. A COMPLETE AND CORRECT DEFINITION OF THE TASK IS GIVEN ON THE NEXT SLIDE.

- BUFFERS ARE NUMBERED 1 TO Total_Buffers. A Boolean ARRAY Available IS MAINTAINED TO KEEP TRACK OF WHICH BUFFERS ARE AVAILABLE. BUFFER i IS AVAILABLE IF AND ONLY IF Available (i) = True. TO OBTAIN A BUFFER, Request IS CALLED. Available IS SEARCHED TO FIND AN AVAILABLE BUFFER, I.E., SOME BUFFER i SUCH THAT Available (i) = True. WHEN THIS IS FOUND, THE BUFFER IS MARKED UNAVAILABLE, BY SETTING ITS Available ENTRY TO False. TO RELEASE A BUFFER, Release IS CALLED. THIS RESULTS IN THE BUFFER'S Available ENTRY TO BE SET TO True.

- THE ATTEMPT SHOWN ON THIS SLIDE FAILS PRECISELY WHEN Request IS SELECTED AND ALL BUFFERS HAVE BEEN ALLOCATED. IN THIS CASE, RATHER THAN LEAVING THE FOR LOOP VIA THE EXIT STATEMENT WITHIN THE IF STATEMENT, THE LOOP IS LEFT NORMALLY AND NO VALUE IS ASSIGNED TO BUFFER. WHEN WE CORRECT THIS, WE WILL ADD A GUARD TO PREVENT Request FROM BEING SELECTED WHEN THERE ARE NO BUFFERS AVAILABLE.

PROBLEM 2 - CONDITIONALLY ACCEPTING AN ENTRY CALL

- A TASK MAY NEED TO GUARD AGAINST ACCEPTING AN ENTRY CALL, OR TO DISTINGUISH BETWEEN TWO CALLS TO THE SAME ENTRY, BUT UNDER DIFFERENT CONDITIONS.
- TASKS OFTEN NEED TO SHARE A FIXED NUMBER OF SOME REUSABLE RESOURCE. THE FOLLOWING CODE FRAGMENT ATTEMPTS TO ALLOW BUFFERS TO BE SHARED.

```
task type Buffer_Allocation_Type is
   entry Request (Buffer : out Buffer_Range_Type);
   entry Release (Buffer : in Buffer_Range_Type);
end Buffer_Allocation_Type;

task body Buffer_Allocation_Type is
   Available : array (1 .. Total_Buffers) of Boolean := (others => True);
begin
   loop
     select
       accept Request (Buffer : out Buffer_Range_Type) do
         Search_Loop:
           for Candidate_Buffer in Resource_Range_Type'Range loop
             if Available (Candidate_Buffer) then
               Available (Candidate_Buffer) := False;
               Buffer := Candidate_Buffer;
               exit Search_Loop;
             end if;
           end loop Search_Loop;
       end Request;
     or
         accept Release (Buffer : in Buffer_Range_Type) do
           Available (Buffer) := True;
       end Release;
     end select;
   end loop;
end Buffer_Allocation_Type;
```

- THIS IS NOT A SOLUTION. WHEN Request IS ACCEPTED WHILE ALL RESOURCES ARE BEING USED, THE SEQUENCE OF STATEMENTS WITHIN THE IF STATEMENT WILL NOT BE EXECUTED.
- WHAT WE REALLY WANT IS TO BE ABLE TO GUARD AGAINST Request BEING ACCEPTED WHEN ALL RESOURCES ARE IN USE, I.E., WE WANT Request TO BE ACCEPTED CONDITIONALLY.

INSTRUCTOR NOTES

- THIS IS A COMPLETE TASK TYPE DEFINITION FOR THE Buffer_Allocation_Type.

- WE ADD A VARIABLE, Number_Of_Buffers_Available, THAT KEEPS TRACK OF Buffers THAT HAVE NOT BEEN ALLOCATED. WE USE THIS TO BUILD A GUARD

  Number_Of_Buffers_Available > 0

  MEANING "DO NOT SELECT THIS ACCEPT STATEMENT *IF THERE ARE NO BUFFERS TO BE GIVEN* OUT."

- NOW WHEN Request IS SELECTED, THERE IS ALWAYS AT LEAST ONE BUFFER THAT IS AVAILABLE. NOW THE LOOP WILL ALWAYS TERMINATE VIA THE EXIT STATEMENT, AND A VALUE WILL BE ASSIGNED TO Buffer.

BUFFER ALLOCATION

```
subtype Buffer_Range_Type is Natural range 1 .. Total_Buffers;

task type Buffer_Allocation_Type is
   entry Request (Buffer : out Buffer_Range_Type);
   entry Release (Buffer : in Buffer_Range_Type);
end Buffer_Allocation_Type;

task body Buffer_Allocation_Type is
   Available : array (Buffer_Range_Type) of Boolean := (others => True);
   Number_Of_Buffers_Available : Natural range 0 .. Total_Buffers := Total_Buffers;
begin -- Buffer_Allocation_Type
  loop
    select
      when Number_Of_Buffers_Available > 0 =>
        accept Request (Buffer : out Buffer_Range_Type) do
          Search_Loop:
            for Candidate_Buffer in Resource_Range_Type'Range loop
              if Available (Candidate_Buffer) then
                Available (Candidate_Buffer) := False;
                Buffer := Candidate_Buffer;
                exit Search_Loop;
              end if;
            end loop Search_Loop;
          Number_Of_Buffers_Available := Number_Of_Buffers_Available - 1;
        end Request;
    or
      accept Release (Buffer : in Buffer_Range_Type) do
        Available (Buffer) := True;
        Number_Of_Buffers_Available := Number_Of_Buffers_Available + 1;
      end Release;
    end select;
  end loop;
end Buffer_Allocation_Type;
```

- EMPHASIZE THAT IF THE SELECTIVE WAIT STATEMENT MUST WAIT FOR AN ENTRY CALL TO BE MADE, THE GUARDS ARE NOT RE-EVALUATED WHEN A CALL IS MADE.

  - GUARDS ARE ONLY EVALUATED AT THE START OF THE STATEMENT.

  - A GUARD CANNOT BECOME TRUE WHILE WAITING.

- POINT OUT THAT CARE MUST BE TAKEN IN SELECTING GUARDS. IF THE GUARD INVOLVES EVALUATION OF A FUNCTION THAT YIELDS DIFFERENT VALUES FOR THE SAME ARGUMENTS, THEN STRANGE RESULTS CAN OCCUR. CONSIDER THE FOLLOWING TWO SELECTIVE WAIT STATEMENTS WHERE Silly IS A Boolean FUNCTION RETURNING THE NEGATION OF ITS PREVIOUS VALUE:

```
select
  when Silly =>
    accept Entry1;
or
  when Silly =>
    accept Entry_2;
end select;

select
  when Silly =>
    accept Entry1;
or
  when not Silly =>
    accept Entry_2;
end select;
```

IN THE SELECTIVE WAIT STATEMENT, ONLY ONE ALTERNATIVE WILL EVER BE OPEN, WHILE IN THE SECOND SELECTIVE WAIT STATEMENT, EITHER BOTH ALTERNATIVES WILL BE OPEN OR NEITHER WILL BE OPEN.

GUARDS

- SELECTIVE WAITS WITH GUARDS ALLOW CONDITIONAL SELECTION OF AN ACCEPT STATEMENT. THIS VERSION OF THE SELECTIVE WAIT HAS THE FORM:

```
          select
        [when  condition   => ]
            accept alternative
        {or
        [when  condition   => ]
            accept alternative }
          end select;
```

THE condition IS CALLED A <u>GUARD</u>.

- AN ALTERNATIVE IS SAID TO BE <u>OPEN</u> IF
    - IT HAS NO GUARD, OR
    - IT HAS A GUARD, AND ITS CONDITION IS TRUE.
- EXECUTION OF A SELECTIVE WAIT STATEMENT
    - FIRST CAUSES ALL GUARDS TO BE EVALUATED,
    - AND THEN PROCEEDS AS WITH THE SIMPLE SELECTIVE WAIT USING ONLY THE OPEN ALTERNATIVES.

INSTRUCTOR NOTES

- THIS IS THE SECOND OF FIVE EXERCISES WITHIN THIS SECTION. ALLOW 20 MINUTES - 15 MINUTES TO DO THE EXERCISE, AND 5 MINUTES TO GO OVER IT.

- THIS EXERCISE IS INTENDED TO DISCOVER WHETHER THOSE STUDENTS THAT WERE NODDING DURING THE PRESENTATION REALLY UNDERSTAND THE MATERIAL.

EXERCISE 8.2 - GUARDS

- MODIFY THE COUNTER TASK TYPE TO MAINTAIN A NONNEGATIVE COUNT.
- THE Increase ENTRY SHOULD NOT BE EXECUTED IF THIS WOULD RESULT IN AN ATTEMPT TO INCREASE THE COUNT PAST Natural'Last.
- THE Decrease ENTRY SHOULD NOT BE EXECUTED IF THIS WOULD RESULT IN AN ATTEMPT TO MAKE THE COUNT GO NEGATIVE.

INSTRUCTOR NOTES

- THIS SLIDE MOTIVATES THE NEED FOR ALLOWING TERMINATE ALTERNATIVES IN SELECT STATEMENTS.

- A TCB IS A TASK CONTROL BLOCK.

PROBLEM 3 - TERMINATING A TASK

- A TASK MAY BE WAITING TO ACCEPT AN ENTRY CALL, BUT OTHER TASKS COMMUNICATING WITH THIS TASK MAY HAVE COMPLETED OR TERMINATED. THIS TASK NEEDS TO BE ABLE TO STOP WAITING AND TERMINATE WHEN THIS HAPPENS.

- THE Shared_Count_Type TASK TYPE CONTAINS A LOOP OF THE FORM:

```
loop
   select
      accept Increase_Count (By : in Positive) do
             ...
      end Increase_Count;

   or

      accept Get_Count (Sum_So_Far : out Natural) do
             ...
      end Get_Count;

   end select;

end loop;
```

- IF THE TASKS USING A Shared_Count_Type OBJECT HAVE COMPLETED, THEN THE Shared_Count_Type OBJECT IS NO LONGER NEEDED. AS WRITTEN, HOWEVER, THE OBJECT WILL STAY AROUND "FOREVER." THIS CAUSES TWO PROBLEMS:

  - THE MASTER OF THIS TASK OBJECT CANNOT TERMINATE
  - RESOURCES USED BY ACTIVE TASKS (SUCH AS STORAGE FOR TCBS) ARE NOT FREED UP.

- WE NEED SOME WAY TO INDICATE THAT THE Shared_Count_Type TASK MAY TERMINATE WHILE WAITING AT THIS POINT.

INSTRUCTOR NOTES

- THIS SLIDE IS JUST THE Shared_Count_Type WITH A TERMINATE ALTERNATIVE.
- NOW IF THE TASKS THAT USE A Shared_Count_Type TASK ARE DONE WITH IT, THEN THE Shared_Count_Type TASK MAY ALSO TERMINATE.

A TERMINATE ALTERNATIVE EXAMPLE - Shared_Count_Type

```
task type Shared_Count_Type is
   entry Increase_Count (By : in Positive);
   entry Get_Count (Sum_So_Far : out Natural);
end Shared_Count_Type;

task body Shared_Count_Type is
   Sum : Natural := 0;
begin -- Shared_Count_Type
   loop
      select
         accept Increase_Count (By : in Positive) do
           declare
             Increment : Positive renames By;
           begin
             Sum := Sum + Increment;
           end; -- block
         end Increase_Count;
      or
         accept Get_Count (Sum_So_Far : out Natural) do
           Sum_So_Far := Sum;
         end Get_Count;
      or
         terminate;
      end select;
   end loop;
end Shared_Count_Type;
```

INSTRUCTOR NOTES

- THIS SLIDE SHOWS THE Buffer_Allocation_Type MODIFIED WITH A TERMINATE ALTERNATIVE.

- WHEN THE TASKS SHARING THE BUFFER POOL, THROUGH A Buffer_Allocation_Type TASK, ARE DONE WITH THE BUFFERS, THEY CAN TERMINATE SINCE THE Buffer_Allocation_Type TASK CAN TERMINATE.

A TERMINATE ALTERNATIVE EXAMPLE - Buffer_Allocation Type

```
task type Buffer_Allocation_Type is
   entry Request (Buffer : out Buffer_Range_Type);
   entry Release (Buffer : in Buffer_Range_Type);
end Buffer_Allocation_Type;

task body Buffer_Allocation_Type is
   Available                  : array (Buffer_Range_Type) of Boolean :=
                                  (others => True);
   Number_Of_Buffers_Available : Natural range 0 .. Total_Buffers := Total_Buffers;
begin -- Buffer_Allocation_Type
   loop
      select
         when Number_Of_Buffers_Available > 0 =>
         accept Request (Buffer : out Buffer_Range_Type) do
            Search_Loop:
            for Candidate_Buffer in Resource_Range_Type'Range loop
               if Available (Candidate_Buffer) then
                  Available (Candidate_Buffer) := False;
                  Buffer := Candidate_Buffer;
                  exit Search_Loop;
               end if;
            end loop Search_Loop;
            Number_Of_Buffers_Available := Number_Of_Buffers_Available - 1;
         end Request;
      or
         accept Release (Buffer : in Buffer_Range_Type) do
            Available (Buffer) := True;
            Number_Of_Buffers_Available := Number_Of_Buffers_Available + 1;
         end Release;
      or
         terminate;

      end select;
   end loop;
end Buffer_Allocation_Type;
```

INSTRUCTOR NOTES

- THIS SLIDE INTRODUCES THE TERMINATE ALTERNATIVE.
- LAST BULLET - THE NEXT SLIDE GOES INTO MORE DETAIL.

THE TERMINATE ALTERNATIVE

- SELECTIVE WAITS WITH A TERMINATE ALTERNATIVE ALLOW A TASK TO TERMINATE IF ITS POTENTIAL CALLERS HAVE TERMINATED. THIS VERSION OF THE SELECTIVE WAIT HAS THE FORM:

```
      select
     [when  condition  => ]
        selective wait alternative
     {or
     [when  condition  => ]
        selective wait alternative }
      end select;
WHERE A  selective wait alternative  IS EITHER
      AN  accept alternative
      OR terminate;
```

- AN OPEN TERMINATE ALTERNATIVE IN TASK T WILL BE SELECTED ONLY IF ALL TASKS THAT MAY COMMUNICATE WITH TASK T HAVE FINISHED. IN THIS CASE, TASK T IS ALSO TERMINATED. THERE MAY ONLY BE ONE TERMINATE ALTERNATIVE.

INSTRUCTOR NOTES

- BULLET 2 - WITHOUT THE TERMINATE ALTERNATIVE, TASK T WOULD WAIT FOREVER, HOLDING ONTO SYSTEM RESOURCES.

- BULLET 3 - GIVEN A TASK T CONTAINING

```
      ...
    accept E_1;
    loop
       select
         accept E_2 ...
       or
         accept E_3 ...
       or
         terminate;
       end select;
    end loop;
```

THE RUNTIME SYSTEM (RTS) WILL NOT ALLOW TASK T TO TERMINATE IF E_2 OR E_3 CAN STILL BE CALLED, OR IF THERE ARE STILL OUTSTANDING (QUEUED) CALLS TO THESE ENTRIES. IF THERE ARE N OUTSTANDING CALLS, THEN THERE ARE N TASKS WAITING TO COMPLETE THEIR RENDEZVOUS WITH TASK T. (FOR THE INSTRUCTOR'S BENEFIT ONLY, THE RTS WILL NOT ALLOW THE TASK TO TERMINATE IF T IS WAITING AT THE TERMINATE ALTERNATIVE, AND THERE IS AN OUTSTANDING CALL TO ENTRY E_1, OR A POTENTIAL CALL. IN EITHER CASE, DEADLOCK WILL OCCUR.)

- BULLET 4 - THE NEXT FEW SLIDES TRY TO EXPLAIN THIS. REMEMBER WE NEVER TALKED ABOUT THE TRANSITIVE RELATION ON MASTERS.

A CLOSER LOOK AT THE TERMINATE ALTERNATIVE

- TASKS CONTAINING INFINITE LOOPS AS IN THE Shared_Count_Type:

```
loop
   select
      accept Increase_Count ... do
       ...
      end Increase_Count;
   or
      accept Get_Count ... do
       ...
      end Get_Count;
   or
      terminate;
   end select;
end loop;
```

OCCUR FREQUENTLY. THEY PERFORM SERVICES FOR OTHER TASKS, AND MUST EXIST AS LONG AS THE TASKS THAT USE THEM.

- THE TERMINATE ALTERNATIVE ALLOWS SUCH A TASK TO TERMINATE WHEN THE TASKS USING IT HAVE FINISHED.

- BEFORE A TERMINATE ALTERNATIVE IN TASK T IS SELECTED, THE RUNTIME SYSTEM ENSURES THAT THERE ARE NO OUTSTANDING CALLS ON TASK T'S ENTRIES, AND THAT IT IS NOT POSSIBLE FOR ADDITIONAL CALLS TO ENTRIES OF TASK T TO BE ISSUED.

- MASTERS MUST BE CONSIDERED TO UNDERSTAND WHEN THE TERMINATE ALTERNATIVE IS SELECTED.

INSTRUCTOR NOTES

- IN THIS SLIDE, AND THE NEXT TWO, NODES AT THE SAME LEVEL REPRESENT TASKS SUCH THAT EITHER

  - THE TASK IS DECLARED IN, OR

  - THE TASK IS ALLOCATED AND ITS ACCESS TYPE DEFINITION IS DEFINED IN

  THE SAME DECLARATIVE REGION, I.E., THE PARENT OF EACH NODE IS ITS DIRECT MASTER.

- IN THIS SLIDE WE SEE THAT A TASK WAITING AT AN OPEN TERMINATE ALTERNATIVE CANNOT HAVE ITS TERMINATE ALTERNATIVE SELECTED IF ANY OF ITS SIBLINGS ARE STILL EXECUTING. WE WILL GO INTO MORE DETAIL IN THE NEXT TWO SLIDES.

- ALSO NOTE THAT IF P HAD NOT COMPLETED THEN T1 WOULD STILL WAIT EVEN IF T2 and T3 TERMINATED.

A CLOSER LOOK AT THE TERMINATE ALTERNATIVE - CONTINUED

LEGEND:

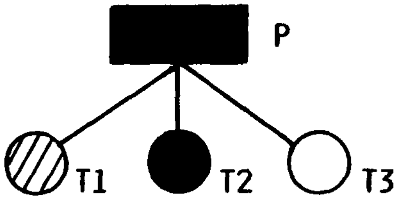
□ MAIN PROGRAM

○ TASK

● - COMPLETED

▨ - WAITING AT TERMINATE ALTERNATIVE

○ - EXECUTING AND NOT WAITING.



- SINCE THE MAIN PROGRAM AND TASK T2 HAVE BOTH COMPLETED, THEY CANNOT ISSUE ENTRY CALLS TO TASK T1. SINCE TASK T3 IS EXECUTING, IT HAS THE POTENTIAL TO ISSUE ENTRY CALLS TO TASK T1. THUS THE TERMINATE ALTERNATIVE IN TASK T1 CANNOT BE SELECTED.

- IF TASK T3 COMPLETES (HENCE TERMINATES) THEN THE TERMINATE ALTERNATIVE IN TASK T1 CAN BE SELECTED, WHICH CAUSES TASK T1 TO TERMINATE. SINCE THE TASKS THAT DIRECTLY DEPEND ON MASTER P HAVE ALL TERMINATED, THIS ALLOWS P TO TERMINATE.

- IF TASK T3 WAITS AT AN OPEN TERMINATE ALTERNATIVE IN A SELECT STATEMENT, THEN IT CANNOT CALL ANY OF TASK T1'S ENTRIES; SIMILARLY, TASK T1 CANNOT CALL ANY OF TASK T3'S ENTRIES. THUS THE TERMINATE ALTERNATIVE OF TASKS T1 AND T3 CAN BE SELECTED SINCE THE TASKS THAT DIRECTLY DEPEND ON MASTER P ARE EITHER TERMINATED (TASK T2) OR CAN HAVE THEIR TERMINATE ALTERNATIVES SELECTED (TASKS T1 AND T2). THUS TASKS T1 AND T2 ARE TERMINATED, WHICH ALLOWS P TO TERMINATE.

INSTRUCTOR NOTES

- BULLET 1 - THE FOLLOWING IS FOR THE INSTRUCTOR'S BENEFIT ONLY. DO NOT PRESENT IN CLASS. SUPPOSE T IS WAITING AT AN OPEN TERMINATE ALTERNATIVE. WE CONSIDER 3 CASES.

  - IF M IS EXECUTING THEN IT CAN CALL ENTRIES OF T, SO T CANNOT TERMINATE.

  - IF M IS COMPLETE, THEN CONSIDER THE FOLLOWING TWO CASES:

    - SOME *OTHER* TASK HAVING M AS A MASTER IS EXECUTING. THIS TASK CAN CAUSE ONE OF T'S ENTRIES TO BE CALLED, SO T CANNOT TERMINATE.

    - ALL TASKS HAVING M AS A MASTER ARE EITHER COMPLETE OR SIMILARLY WAITING AT OPEN TERMINATE ALTERNATIVES. THE ENTRIES OF TASK T CANNOT BE CALLED OUTSIDE OF M SO T MAY TERMINATE, ALONG WITH M AND ALL THE TASKS HAVING M AS A MASTER. (NOTE THAT AT THIS POINT ALL OF THE CONDITIONS OF BULLET 1 ARE SATISFIED.)

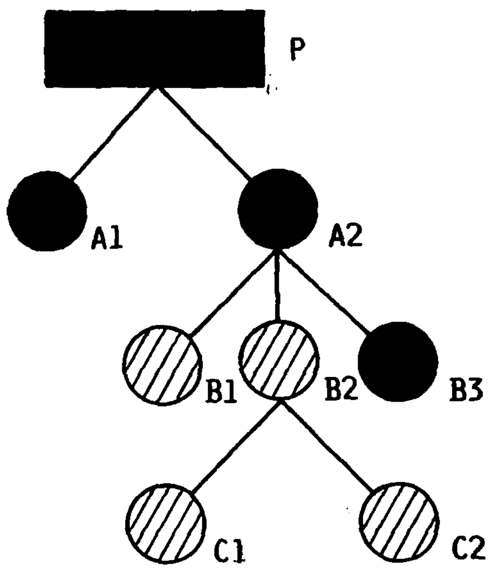
  - IF M IS A TASK WAITING AT AN OPEN TERMINATE ALTERNATIVE, THEN CONSIDER THE FOLLOWING TWO CASES:

    - SOME TASK A, HAVING M AS A MASTER, IS STILL EXECUTING. SINCE A CAN STILL CALL ONE OF T'S ENTRIES, T CANNOT TERMINATE.

    - ALL TASKS HAVING M AS A MASTER ARE EITHER COMPLETE OR WAITING AT AN OPEN TERMINATE ALTERNATIVE. LET X BE THE MASTER OF M. REPEAT THE ANALYSIS IN THIS BULLET USING M AND X FOR T AND M, RESPECTIVELY. (SINCE M IS A TASK, IT MUST HAVE A MASTER.)

- BULLET 2 - SHOW HOW THESE TASKS SATISFY THE CONDITIONS IN BULLET 1. ALSO TAKE THE CLASS THROUGH THIS FIGURE WITH TASK B1 EXECUTING INSTEAD OF WAITING. SHOW THEM WHY THE OTHER WAITING TASKS, A2 AND P CANNOT TERMINATE IF B1 IS EXECUTING.

A CLOSER LOOK AT THE TERMINATE ALTERNATIVE - CONCLUSION



- AN OPEN TERMINATE ALTERNATIVE OF A TASK T CAN BE SELECTED IF IT HAS A MASTER M SUCH ᵀ4AT:
  - M HAS COMPLETED, AND
  - EACH TASK HAVING M AS A MASTER IS EITHER
    - A COMPLETED TASK, OR
    - A TASK WAITING AT AN OPEN TERMINATE ALTERNATIVE.
- TASK A2 PLAYS THE ROLE OF M FOR TASKS B1, B2, C1 AND C2.

INSTRUCTOR NOTES

- THIS IS THE THIRD OF FIVE EXERCISES WITHIN THIS SECTION. ALLOW 10 MINUTES - 5 MINUTES TO DO THE EXERCISE, AND 5 MINUTES TO GO OVER IT.

- THIS EXERCISE IS INTENDED TO DISCOVER WHETHER THOSE STUDENTS THAT WERE NODDING DURING THE PRESENTATION REALLY UNDERSTAND THE MATERIAL.

EXERCISE 8.3 - THE TERMINATE ALTERNATIVE

- MODIFY THE COUNTER TASK TYPE AS FOLLOWS:
  - ALLOW THE COUNTER TASK TO TERMINATE WHEN TASKS USING IT ARE FINISHED WITH IT.
  - ALLOW TASKS USING THE COUNTER TASK TO TERMINATE WHEN THEY ARE DONE WITH IT.

INSTRUCTOR NOTES

- THIS SLIDE EXPLAINS WHY TASKS MUST BE ABLE TO DELAY THEMSELVES.

- AFTER GIVING THIS MOTIVATION, WE TALK ABOUT THE delay STATEMENT AND THE PREDEFINED TYPE Duration.

- FINALLY, WE TALK ABOUT THE PREDEFINED PACKAGE Calendar AND THE OPERATIONS ON Duration OBJECTS.

- BULLET 1 - IN ORDER TO TEST A RADAR SYSTEM, ROCKET LAUNCHES ARE SIMULATED. THE TIME BETWEEN LAUNCHES MUST BE AT LEAST SOME FIXED DURATION.

- BULLET 2 - AS A SATELLITE PROGRESSES ALONG ITS TRAJECTORY, IT MAY LOSE CONTACT WITH THE GROUND STATION (IN THE FIGURE, THE LOSS OCCURS AT POINT A). RATHER THAN BUSY WAITING, THE TASK CALCULATES THE TIME UNTIL IT CAN ACQUIRE A SIGNAL FROM THE NEXT GROUND STATION (AT POINT B IN THE FIGURE). THE TASK DELAYS ITSELF UNTIL THEN.

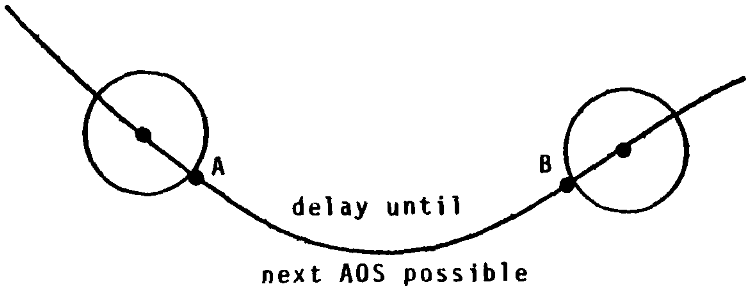
PROBLEM 4 - DELAYING A TASK

- A TASK MIGHT WANT TO DELAY ITSELF BECAUSE IT IS PART OF A REAL-TIME SIMULATION, SAY, OF A RADAR SYSTEM:

```
for I in 1 .. Number_Of_Rockets loop
   Simulate_Rocket_Launch;
   -- Delay for time between launches
end loop;
```

- A TASK MIGHT WANT TO DELAY ITSELF TO AVOID <u>BUSY</u> <u>WAITING</u>, SAY, IN A SATELLITE TELEMETRY TRANSMISSION TASK:

```
-- Compute next expected AOS - (Acquisition Of Signal)
-- Delay until then
-- Try to establish communications
```



- A TASK MIGHT WANT TO DELAY ITSELF IF IT IS TO POLL FOR SOME EVENT BUT DOES NOT WANT TO WASTE CPU TIME BY POLLING TOO OFTEN.

  - SAMPLING TEMPERATURE
  - POLLING FOR KEYSTROKES

INSTRUCTOR NOTES

- THIS SLIDE INTRODUCES THE delay STATEMENT AND THE PREDEFINED TYPE Duration.

- OPERATIONS ON Duration OBJECTS ARE DESCRIBED IN THE NEXT SLIDE DEALING WITH Calendar.

- NOTE THAT IMPLEMENTATIONS ARE ENCOURAGED TO USE 50 MICROSECONDS OR LESS AS THE SMALLEST REPRESENTABLE Duration.

- Duration MAY OR MAY NOT BE REPRESENTED INTERNALLY IN THE SAME WAY AS THE SYSTEM CLOCK.

- BULLET 3 - TELL THE CLASS THAT DIFFICULTIES CAN OCCUR BECAUSE THE DELAY IS NOT EXACT. ALSO TELL THEM THAT WE WILL ADDRESS THESE DIFFICULTIES IN SECTION 14 (CYCLIC PROCESSING).

Delay STATEMENTS AND TYPE Duration

- WHEN A TASK WANTS TO DELAY ITSELF IT USES THE DELAY STATEMENT:

```
delay expression ;
```

  WHERE expression IS A SIMPLE EXPRESSION OF TYPE Duration.

- Duration IS A PREDEFINED FIXED POINT TYPE WHOSE VALUES EXPRESS SECONDS.
- THE DELAY STATEMENT DELAYS THE TASK FOR AT LEAST THE DURATION SPECIFIED.
  - THE RUNTIME SYSTEM MIGHT NOT ALLOCATE THE CPU TO A TASK THE INSTANT THAT ITS DELAY EXPIRES, SO THE EFFECTIVE DELAY CAN BE LONGER.
- THE DELAY STATEMENT TREATS A NEGATIVE DURATION AS A ZERO DURATION.
- OPERATIONS ON OBJECTS IN TYPE Duration ARE DEFINED IN THE PREDEFINED PACKAGE Calendar.
- EXAMPLES:

```
delay 5.0; -- delay for 5 seconds

delay 0.5; -- delay for 1/2 second
```

- AN IMPLEMENTATION PROVIDES FOR Duration VALUES OF AT LEAST - 86,400 SECONDS TO + 86,400 SECONDS (ONE DAY).
- THE SMALLEST REPRESENTABLE DURATION (Duration'Small) AND THE ACTUAL Duration RANGE IS SPECIFIED IN Appendix F.

INSTRUCTOR NOTES

- THE Calendar PACKAGE HAS NOT BEEN INTRODUCED IN EARLIER COURSES. IT IS BEING INTRODUCED HERE PRIMARILY TO ALLOW US TO TALK ABOUT Duration EXPRESSIONS.

- IN THIS SLIDE TALK ABOUT THE OPERATIONS ON DURATION AND ABOUT Time AND Clock.

- ALSO TELL THE CLASS THAT ALL OF THE USUAL FIXED POINT ARITHMETIC OPERATIONS ARE AVAILABLE ON THE TYPE Duration.

- MAKE SURE THE CLASS UNDERSTANDS THE DIFFERENCE BETWEEN Time (A POINT ON A TIME LINE) AND Duration (THE DISTANCE BETWEEN TWO POINTS ON A TIME LINE).

THE Calendar PACKAGE

```
package Calendar is
   type Time is private;

   subtype Year_Number  is Integer  range 1901 .. 2099;
   subtype Month_Number is Integer  range 1 .. 12;
   subtype Day_Number   is Integer  range 1 .. 31;
   subtype Day_Duration is Duration range 0.0 .. 86_400.0; -- One Day

   function Clock return Time;

   function Year    (Date : Time) return Year_Number;
   function Month   (Date : Time) return Month_Number;
   function Day     (Date : Time) return Day_Number;
   function Seconds (Date : Time) return Day_Duration;

   procedure Split (Date    : in Time;
                    Year    : out Year_Number;
                    Month   : out Month_Number;
                    Day     : out Day_Number;
                    Seconds : out Day_Duration);

   function Time_Of (Year    : Year_Number;
                     Month   : Month_Number;
                     Day     : Day_Number;
                     Seconds : Day_Duration := 0.0) return Time;

   function "+"  (Left : Time;     Right : Duration) return Time;
   function "+"  (Left : Duration; Right : Time)     return Time;
   function "-"  (Left : Time;     Right : Duration) return Time;
   function "-"  (Left : Time;     Right : Time)     return Duration;

   function "<"  (Left : Time;     Right : Time)     return Boolean;
   function "<=" (Left : Time;     Right : Time)     return Boolean;
   function ">"  (Left : Time;     Right : Time)     return Boolean;
   function ">=" (Left : Time;     Right : Time)     return Boolean;

   Time_Error : exception; -- can be raised by Time_Of, "+", and "-"

private -- implementation dependent
end Calendar;
```

INSTRUCTOR NOTES

- THIS SLIDE REVISITS THE TWO EXAMPLES OF TASKS DELAYING THEMSELVES.

- POINT OUT THE USE OF Clock.

- THE EXPRESSION Time_Of_Next_AOS - Calendar.Clock SUBTRACTS TWO OBJECTS OF TYPE Time AND YIELDS AN EXPRESSION OF TYPE Duration.

DELAY STATEMENT EXAMPLES

EXAMPLE:

```
for I in 1 .. Number_Of_Rockets loop
   Simulate_Rocket_Launch;
   delay Time_Between_Launches;
end loop;
```

EXAMPLE:

```
         ...
  Time_Of_Next_AOS : Time;
         ...
begin
         ...
  Time_Of_Next_AOS := Compute_Next_Expected_AOS;
  delay Time_Of_Next_AOS - Calendar.Clock;
  Establish_Communications;
         ...
end;
```

INSTRUCTOR NOTES

- THIS SLIDE MOTIVATES THE NEED FOR A DELAY ALTERNATIVE.

PROBLEM 5 - WHEN A TASK'S ENTRIES ARE NOT CALLED IN TIME

- IF A TASK'S ENTRIES HAVE NOT BEEN CALLED WITHIN SOME SPECIFIED TIME, THE TASK MAY NEED TO TAKE CORRECTIVE ACTION OR RAISE AN ALARM.

- A TASK, IN A NAVIGATION SYSTEM, ACCEPTS AN ENTRY TO REPORT POSITION AND VELOCITY. THE TASK IN TURN DISPLAYS THE CURRENT POSITION. THE TASK MIGHT CONTAIN THE CODE FRAGMENT:

```
loop

   accept Report_Navigation_Data
             (Velocity : in Velocity_Type;
              Position : in Position_Type) do

       Current_Velocity := Velocity;
       Current_Position := Position;

   end Report_Navigation_Data;

   Display_Position (Current_Position);

end loop;
```

- NOW WHAT HAPPENS IF A NEW POSITION DOES NOT ARRIVE "IN TIME", I.E., IF THE POSITION GETS OLD? USING THE OLD POSITION MAY BE MISLEADING.

- WHAT WE WANT TO BE ABLE TO DO IS CALCULATE A NEW POSITION BY EXTRAPOLATION IF A NEW POSITION IS NOT DELIVERED IN TIME.

INSTRUCTOR NOTES

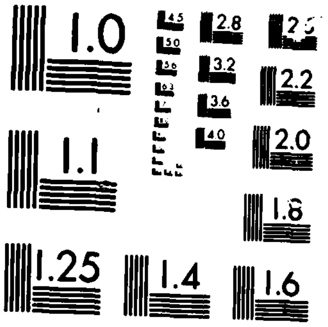
- THIS SLIDE SHOWS HOW THE PROBLEM CAN BE SOLVED.

- THE FUNCTION Projected_Position CALCULATES A NEW POSITION VIA EXTRAPOLATION. IT WILL BE USED IF AN UPDATED POSITION IS NOT RECEIVED IN TIME.

- FOR THE INSTRUCTOR'S BENEFIT, THE PROBLEM OF CUMULATIVE DRIFT - DISCUSSED IN *SECTION 14 - IS NOT A PROBLEM HERE. CALLS TO Report_Navigation_Data* OCCUR MUCH MORE FREQUENTLY THAN EXTRAPOLATION. THE CALLS EFFECTIVELY SYNCHRONIZE THE LOOP. WELL BEFORE WE COULD EVEN NOTICE CUMMULATIVE DRIFT, WE WOULD NEED TO HAVE A SITUATION WHERE Report_Navigation_Data WAS NOT BEING CALLED - A FAR MORE SERIOUS PROBLEM.

AD-A166 351 ADA (TRADE NAME) TRAINING CURRICULUM REAL-TIME SYSTEMS 4/5
IN ADA L401 TEACHER'S GUIDE VOLUME 1(U) SOFTECH INC
WALTHAM MA 1986 DAAB07-83-C-K514

UNCLASSIFIED F/G 5/9 NL



MICROCOPY RESOLUTION TEST CHART

DELAY ALTERNATIVE EXAMPLE - NAVIGATION DATA EXTRAPOLATION

- A delay STATEMENT CAN APPEAR AT THE BEGINNING OF A SELECTIVE WAIT ALTERNATIVE.
- THE delay STATEMENT HAS A DIFFERENT MEANING IN THIS CONTEXT.

```
loop
   select
      accept Report_Navigation_Data
              (Velocity : in Velocity_Type;
               Position : in Position_Type) do
         Current_Velocity := Velocity;
         Current_Position := Position;
      end Report_Navigation_Data;
   or
      delay 0.5;
      Current_Position := Projected_Position
                           (Current_Position,
                            Current_Velocity);
   end select;
   Display_Position (Current_Position);
end loop;
```

- EACH EXECUTION OF THE SELECTIVE WAIT WILL EITHER
  - ACCEPT A CALL ON Report_Navigation_Data, OR
  - AFTER WAITING 0.5 SECONDS WITHOUT SUCH A CALL ARRIVING, EXECUTE THE BOTTOM ASSIGNMENT STATEMENT.

INSTRUCTOR NOTES

- THIS SLIDE INTRODUCES THE DELAY ALTERNATIVE.

- MAKE SURE THE STUDENTS UNDERSTAND THE DIFFERENCE BETWEEN THE USE OF THE DELAY STATEMENT AS A SIMPLE STATEMENT AND THE DELAY STATEMENT AS A DELAY ALTERNATIVE. IT IS PROBABLY BETTER IF WE CONSISTENTLY REFER TO THIS LATER USE OF THE DELAY STATEMENT AS A DELAY ALTERNATIVE. THIS SHOULD ELIMINATE THE POSSIBLE CONFUSION BETWEEN THE TWO DISTINCT USES OF DELAYS.

THE DELAY ALTERNATIVE

- SELECTIVE WAITS WITH DELAY ALTERNATIVES ALLOW A TASK TO PERFORM ALTERNATIVE ACTIONS IF ITS ENTRIES ARE NOT CALLED IN A REASONABLE - FOR THE TASK SPECIFIED - TIME. THIS VERSION OF THE SELECTIVE WAIT STATEMENT HAS THE FORM:

```
 select
[when condition => ]
     selective wait alternative
{or
[when condition => ]
     selective wait alternative }
end select;
```

WHERE A selective wait alternative IS EITHER

```
AN accept alternative
OR delay expression;
  [ sequence of statements ]
```

- AN OPEN DELAY ALTERNATIVE WILL BE SELECTED ONLY IF NO ACCEPT ALTERNATIVE CAN BE SELECTED BEFORE THE SPECIFIED DELAY HAS ELAPSED. IF SEVERAL DELAY ALTERNATIVES ARE OPEN, THE ONE WITH THE SHORTEST DELAY IS SELECTED - TIES ARE RESOLVED ARBITRARILY.

INSTRUCTOR NOTES

- THIS IS THE FOURTH OF FIVE EXERCISES WITHIN THIS SECTION. ALLOW 30 MINUTES - 20 MINUTES TO DO THE EXERCISE, AND 10 MINUTES TO GO OVER IT.

- THIS EXERCISE IS INTENDED TO DISCOVER WHETHER THOSE STUDENTS THAT WERE NODDING DURING THE PRESENTATION REALLY UNDERSTAND THE MATERIAL.

EXERCISE 8.4 - THE DELAY ALTERNATIVE

- MODIFY THE COUNTER TASK TYPE AS FOLLOWS:
  - REMOVE THE TERMINATE ALTERNATIVE
  - THE COUNTER TASK TYPE SHOULD PERFORM THE FOLLOWING ACTION BASED ON THE VALUE OF THE COUNTER.
    - IF THE COUNTER IS GREATER THAN Natural'Last/2 + 100 THEN IF NO ENTRY IS CALLED WITHIN 5 SECONDS, SET THE COUNTER TO Natural'Last.
    - IF THE COUNTER IS LESS THAN Natural'Last/2 - 100 THEN IF NO ENTRY IS CALLED WITHIN 15 SECONDS, SET THE COUNTER TO 0.
    - OTHERWISE, IF NO ENTRY IS CALLED WITHIN 30 SECONDS, THEN SET THE COUNTER TO Natural'Last/2.

INSTRUCTOR NOTES

- THIS SLIDE PROVIDES MOTIVATION FOR ELSE PARTS IN SELECTIVE WAIT STATEMENTS.
- EACH TIME THROUGH THE LOOP, Change_Light_If_Necessary IS CALLED TO CHANGE THE CHECK IF THE LIGHT NEEDS TO BE CHANGED AND TO CHANGE IT IF IT SHOULD BE.
- THE POLL IS TO OCCUR EVERY SECOND. WE INCLUDE THE POLL TO AVOID BUSY WAITING.

PROBLEM 6 - ACCEPTING URGENT ENTRY CALLS

- SOMETIMES A TASK WILL WANT TO REPETITIVELY PERFORM SOME ROUTINE WORK UNLESS SOME EXTRAORDINARY EVENT OCCURS.

- A TASK FOR TRAFFIC LIGHTS MIGHT CHANGE LIGHTS PERIODICALLY. THE TASK MIGHT CONTAIN A LOOP SUCH AS:

```
loop

   Change_Light_If_Necessary;

   delay Next_Poll_Time - Calendar.Clock;
   Next_Poll_Time := Calendar.Clock + 1.0;

end loop;
```

- IF A CITY DECIDES TO INSTALL TRAFFIC LIGHTS ALONG A MAJOR ROAD AND ALLOW EMERGENCY VEHICLES (FIRE, POLICE, AMBULANCE, MY MOTHER'S CAR) TO PROCEED ALONG THE ROAD WITHOUT INTERFERENCE FROM SIDE STREETS, WE WOULD NEED TO MODIFY THE TASK TO HANDLE THIS.

- WHAT WE NEED IS A WAY TO HANDLE REGULAR TRAFFIC LIGHT CONTROL AS THE NORMAL CASE, AND EMERGENCY VEHICLE TRAFFIC AS AN URGENT ENTRY CALL.

INSTRUCTOR NOTES

- THIS SLIDE SOLVES THE PREVIOUS PROBLEM.

- NORMALLY, THE ELSE PART OF THE SELECTIVE WAIT STATEMENT IS EXECUTED. THIS SIMPLY CHANGES THE LIGHT IF NECESSARY.

- WHEN THE URGENT ENTRY CALL Start_Emergency_Light_Pattern IS ISSUED, THE ACCEPT ALTERNATIVE IS SELECTED INSTEAD. THE TRAFFIC LIGHT IS PLACED IN THE EMERGENCY LIGHT PATTERN, AND THE TASK WAITS UNTIL NORMAL TRAFFIC IS TO RESUME (WHEN End_Emergency_Light_Pattern IS CALLED).

- IN EITHER CASE THE LOOP IS DELAYED UNTIL THE NEXT POLL TIME.

SELECTIVE WAIT WITH AN ELSE PART - EXAMPLE

```
loop

   select

       accept Start_Emergency_Light_Pattern do
          Set_Emergency_Light_Pattern;
       end Start_Emergency_Light_Pattern;

       accept End_Emergency_Light_Pattern;
       Restart_Lights;

   else

       Change_Light_If_Necessary;

   end select;

   delay Next_Poll_Time - Calendar.Clock;
   Next_Poll_Time := Next_Poll_Time + 1.0;

end loop;
```

INSTRUCTOR NOTES

- THIS SLIDE DESCRIBES SELECTIVE WAITS WITH ELSE PARTS.

SELECTIVE WAITS WITH ELSE PARTS

- SELECTIVE WAITS WITH ELSE PARTS ALLOW A TASK TO DO ALTERNATIVE (OR NORMAL) PROCESSING WHEN ITS ENTRIES HAVE NOT BEEN CALLED. THIS VERSION OF THE SELECTIVE WAIT HAS THE FORM:

```
   select
  [when condition  => ]
      selective wait alternative
  {or
  [when condition  => ]
      selective wait alternative }
   else
      sequence of statements
   end select;
```

- THE ELSE PART IS SELECTED IF NO ACCEPT ALTERNATIVE CAN BE IMMEDIATELY ACCEPTED (OR THERE ARE NO OPEN ACCEPT ALTERNATIVES).

INSTRUCTOR NOTES

- THIS IS THE LAST OF THE FIVE EXERCISES WITHIN THIS SECTION. ALLOW 20 MINUTES - 15 MINUTES TO DO THE EXERCISE, AND 5 MINUTES TO GO OVER IT.

- THIS EXERCISE IS INTENDED TO DISCOVER WHETHER THOSE STUDENTS THAT WERE NODDING DURING THE PRESENTATION REALLY UNDERSTAND THE MATERIAL.

EXERCISE 8.5 - THE ELSE PART

- REMOVE THE DELAY ALTERNATIVE.

- MODIFY THE COUNTER TASK TYPE AS FOLLOWS:

  - THE TASK SHOULD LOOK FOR REQUESTS ONLY ONCE EACH SECOND.

  - A REQUEST TO UPDATE THE COUNTER MAY ONLY BE ACCEPTED IF THERE IS NO REQUEST TO READ THE COUNTER.

- <u>HINT</u>: TREAT A REQUEST TO READ THE COUNTER AS MORE URGENT THAN THE OTHER REQUESTS.

INSTRUCTOR NOTES

- THIS SLIDE, AND THE NEXT TWO, SUMMARIZE THE SELECTIVE WAIT STATEMENT.

- THIS SLIDE GIVES THE COMPLETE SYNTAX FOR A SELECTIVE WAIT STATEMENT.

PUTTING IT ALL TOGETHER - SELECTIVE WAIT SYNTAX

```
          select
         [when  condition  =>]
              selective wait alternative
         {or
         [when  condition  => ]
              selective wait alternative }
         [else
              sequence of statements  ]
          end select;
WHERE A  selective wait alternative  HAS THE FORM:
                   accept statement
                   sequence of statements
OR
          delay  expression  ;
                   sequence of statements
OR
          terminate;
```

INSTRUCTOR NOTES

- THIS SLIDE GIVES THE BASIC RULES AND DEFINITIONS.

- AFTER GOING OVER THE RULES ON GUARDS, GO THROUGH THIS EXAMPLE USING DIFFERENT VALUES FOR X. THIS WILL SHOW VARIOUS COMBINATIONS OF OPEN ALTERNATIVES. CONSIDER USING 0, 50, 100 AND 400.

PUTTING IT ALL TOGETHER - SELECTIVE WAIT RULES AND DEFINITIONS

- THERE MUST BE AT LEAST ONE ACCEPT ALTERNATIVE IN A SELECTIVE WAIT.
- IN ADDITION, AT MOST ONE OF THE FOLLOWING MAY ALSO APPEAR:
  - A TERMINATE ALTERNATIVE
  - ONE OR MORE DELAY ALTERNATIVES
  - AN ELSE PART.
- A SELECT ALTERNATIVE IS OPEN IF EITHER
  - IT DOES NOT BEGIN WITH A GUARD, OR
  - IT DOES BEGIN WITH A GUARD AND THE GUARD EVALUATES TO TRUE.

  OTHERWISE THE SELECT ALTERNATIVE IS CLOSED.

```
select
   when X < 500 =>
       accept One;
       X := X + 1;
or
   when X > 0 =>
       accept Two;
       X := X - 1;
or
       accept Three;
or
   when X > 200 =>
       delay 5.0;
       X := Positive'Last;
or
   when X < 100 =>
       delay 15.0;
       X := 0;
or
       delay 30.0;
       X := 0;
end select;
```

INSTRUCTOR NOTES

- THIS SLIDE EXPLAINS THE STEPS IN EXECUTING A SELECTIVE WAIT.

PUTTING IT ALL TOGETHER - SELECTIVE WAIT EXECUTION

- BEFORE ANY ALTERNATIVES ARE CONSIDERED:
  - GUARDS ARE EVALUATED (IN AN UNSPECIFIED ORDER). THIS DETERMINES THE OPEN ALTERNATIVES.
  - ANY OPEN DELAY ALTERNATIVES HAVE THEIR DELAY EXPRESSIONS EVALUATED.
- OPEN ACCEPT ALTERNATIVES ARE CONSIDERED FIRST:
  - IF ANY OF THE ACCEPT STATEMENTS CAN BE IMMEDIATELY SELECTED FOR RENDEZVOUS, THEN ONE OF THE CORRESPONDING ACCEPT ALTERNATIVES IS SELECTED ARBITRARILY. (THIS INCLUDES THE CASE WHERE SEVERAL OPEN ALTERNATIVES EXIST FOR THE SAME ENTRY.)
  - OTHERWISE, IF AN ELSE PART DOES NOT EXIST, THE TASK WAITS UNTIL AN OPEN ACCEPT STATEMENT CAN BE SELECTED.
- IF THE SELECTIVE WAIT HAS ONE OR MORE OPEN DELAY ALTERNATIVES, THEN ONE IS SELECTED IF AND ONLY IF AN ACCEPT ALTERNATIVE CANNOT BE SELECTED WITHIN THE SPECIFIED DELAY.
  - IF SEVERAL OPEN DELAY ALTERNATIVES EXIST, THEN ONE WITH THE SMALLEST DELAY IS SELECTED. IF MORE THAN ONE HAS THIS SAME DELAY, THEN ONE IS CHOSEN ARBITRARILY.
  - NEGATIVE DELAYS ARE TREATED AS ZERO DELAYS.
- IF THE SELECTIVE WAIT HAS AN ELSE PART, THEN IT IS SELECTED IF AND ONLY IF NO ACCEPT STATEMENT CAN BE SELECTED IMMEDIATELY.
  - THIS INCLUDES THE CASE WHERE NO OPEN ALTERNATIVES EXIST.
  - IF ALL ALTERNATIVES ARE CLOSED AND THERE IS NO ELSE PART, THEN THE EXCEPTION Program_Error IS RAISED.

INSTRUCTOR NOTES

PUTTING IT ALL TOGETHER - SELECTIVE WAIT EXECUTION (CONTINUED)

- IF THE SELECTIVE WAIT HAS AN OPEN TERMINATE ALTERNATIVE, THEN IT IS SELECTED IF AND ONLY IF THE TASK CONTAINING THE TERMINATE ALTERNATIVE HAS A MASTER M SUCH THAT

  - M IS COMPLETE.

  - ANY TASK HAVING M AS A MASTER IS EITHER

    - COMPLETE, OR

    - SIMILARLY WAITING AT AN OPEN TERMINATE ALTERNATIVE.

  IN THIS CASE, M AND ALL TASKS HAVING M AS A MASTER TERMINATE.

INSTRUCTOR NOTES

- THIS EXAMPLE SHOWS THE CLASS HOW THE VARIOUS FORMS OF THE SELECTIVE WAIT STATEMENT CAN BE COMBINED.

  THE EXAMPLE CAN BE SKIPPED IF THE INSTRUCTOR IS SHORT ON TIME. COMPLETE COVERAGE REQUIRES ABOUT 25 MINUTES.

EXAMPLE - CONTROLLING A TRAFFIC LIGHT

- DURING NORMAL OPERATION
  - A TRAFFIC LIGHT MUST ENSURE THE FOLLOWING:
    - THERE MUST BE AT LEAST 30 SECONDS BETWEEN LIGHT CHANGES.
    - A LIGHT CHANGE MUST OCCUR AFTER 3 MINUTES.
    - DURING EACH LIGHT CHANGE CYCLE, THE STREET WITH THE HEAVIEST TRAFFIC VOLUME IN THE LAST 10 MINUTES SHOULD HAVE THE GREEN LIGHT FOR A PROPORTIONATELY LONGER TIME.
  - A PEDESTRIAN MAY DEPRESS A CROSSING BUTTON TO FORCE A LIGHT CHANGE. (PUSHING THE BUTTON TAKES EFFECT AS SOON AS THE 30 SECOND MINIMUM INTERVAL BETWEEN LIGHT CHANGES HAS EXPIRED, OR IMMEDIATELY IF THE INTERVAL HAS ALREADY EXPIRED.)
    - TO ALLOW EMERGENCY VEHICLES TO PASS SAFELY AND RAPIDLY, THE TRAFFIC LIGHT CAN BE PUT INTO AN EMERGENCY LIGHT PATTERN.

INSTRUCTOR NOTES

- WE ASSUME THAT THE Traffic_Package IS AVAILABLE TO HELP US CONTROL THE TRAFFIC LIGHT. WE WILL NOT CONSIDER HOW THE THREE PROCEDURES ACCOMPLISH THEIR GOALS.

- WE WILL LOOK AT THE Traffic_Flow_Package IN DETAIL.

THE Traffic_Package AND Traffic_Flow_Package

- THE FOLLOWING PACKAGE IS AVAILABLE:

```
package Traffic_Package is

      procedure Start_Light;
        -- Called when the traffic light is started up or is being
        --   restarted after an emergency traffic light pattern.
      procedure Set_Emergency_Light_Pattern;
        -- Sets the traffic light to the emergency traffic light pattern.
        --   Start_Light must be called to put the traffic light back into
        --   normal operation.
      procedure Change_Light;
        -- Changes the light in both directions.

end Traffic_Package;
```

- WE WILL DEVELOP THE FOLLOWING PACKAGE:

```
package Traffic_Flow_Package is

      task type Traffic_Flow_Task_Type is
           -- Monitors traffic flow along one street
         entry Report_Vehicle_Passage;
           -- Called each time a vehicle passes through the intersection
           --   along this street.
         entry Obtain_Traffic_Flow (Passage_Count : out Natural);
           -- Yields the number of vehicles that have passed through the
           --   intersection along this street in the last 10 minutes.

      end Traffic_Flow_Task_Type;

end Traffic_Flow_Package;
```

INSTRUCTOR NOTES

- THE DISCARD QUEUE IS USED TO NOTE THE OCCURRENCE OF THE EVENT "A VEHICLE HAS PASSED BY." THE EVENT IS NOTED BY PLACING THE TIME OF THE EVENT ON THE QUEUE. (THE EVENT IS NOTED BY CALLING THE Report_Vehicle_Passage ENTRY. THIS IS CALLED AS THE RESULT OF A SENSOR DETECTING A VEHICLE. WE ARE NOT INTERESTED IN HOW THIS IS DONE.) WHEN THE QUEUE IS NOT EMPTY (NOTE THE GUARD) THE DELAY ALTERNATIVE IS USED TO DETECT THE EVENT "A VEHICLE PASSAGE HAS BEEN NOTED FOR 10 MINUTES OR MORE." THIS HAPPENS WHEN THE DELAY EXPRESSION GOES TO ZERO, OR THE TASK HAS BEEN DELAYED FOR THE DURATION SPECIFIED BY THE DELAY EXPRESSION. IN EITHER CASE, THIS CAUSES THE HEAD OF THE QUEUE TO BE REMOVED.

- THE Obtain_Traffic_Flow ENTRY IS CALLED TO FIND OUT HOW MANY VEHICLES HAVE PASSED IN THE LAST 10 MINUTES. THE QUEUE LENGTH GIVES US THIS INFORMATION.

- DO NOT GET INTO DETAILS ABOUT QUEUES!

- IF SOMEONE ASKS, MENTION THAT THIS SOLUTION ASSUMES OPEN ALTERNATIVES WILL BE ACCEPTED IN A FAIR MANNER, I.E., EVEN IF TRAFFIC FLOW IS HEAVY (EVEN IF THERE ARE MANY CALLS TO Report_Car_Passage), SOME CALLS TO Obtain_Traffic_Flow WILL BE ACCEPTED, AS WILL THE DELAY ALTERNATIVE.

Traffic_Flow_Task_Type

```
with Calendar, Generic_Queue_Package;
package body Traffic_Flow_Package is
   task body Traffic_Flow_Task_Type is
      use Calendar;
      package Time_Queue_Package is new Generic_Queue_Package (Time);
      Discard_Queue      : Time_Queue_Package.Queue_Type;
      In_Queue_Duration : constant := 10.0 * 60; -- 10 minutes
      Item               : Time;
   begin -- Traffic_Flow_Task_Type
      loop
         select
            accept Report_Car_Passage;
                  -- Note the passage of this vehicle on the queue.  Keep it there
                  --   for 10 minutes.
            Time_Queue_Package.Enqueue (Discard_Queue, Clock + In_Queue_Duration);
         or
            accept Obtain_Traffic_Flow (Passage_Count : out Natural) do
                  -- The queue contains an entry for each vehicle that has passed
                  --   through the intersection along this street in the last 10
                  --   minutes.  Therefore, the queue size is the count of the
                  --   number of vehicles that have passed.
               Passage_Count := Time_Queue_Package.Queue_Size (Discard_Queue);
            end Obtain_Traffic_Flow;
         or
            when Time_Queue_Package.Queue_Size (Discard_Queue) > 0 =>
                  -- If the passage of this vehicle has been noted for 10 minutes,
                  --   then remove it from the queue.
               delay Time_Queue_Package.Front_Item (Ciscard_Queue) - Clock;
               Time_Queue_Package.Dequeue (Discard_Queue, Item);
         end select;
      end loop;
   end Traffic_Flow_Task_Type;
end Traffic_Flow_Package;
```

INSTRUCTOR NOTES

- WE ASSUME THAT ONE OR MORE INTERFACE TASKS ARE CALLING THE ENTRIES OF THE Traffic_Control_Task, BUT WE DO NOT CARE ABOUT THAT TASK.

Traffic_Control_Package - Specification

```
package Traffic_Control_Package is

   task Traffic_Control_Task is

      entry Start_Emergency_Light_Pattern;
        -- Called to put the light in the emergency light pattern.

      entry End_Emergency_Light_Pattern;
        -- Called to allow light to return to normal operation.

      entry Crossing_Button;
        -- Called when a Pedestrian has depressed the crossing button.
        --   This usually occurs when a Pedestrian accuses it of not
        --   responding fast enough.
   end Traffic_Control_Task;

end Traffic_Control_Package;
```

INSTRUCTOR NOTES

- Min_Change_Time AND Max_Change_Time ARE THE MIN AND MAX TIMES THAT A LIGHT CAN STAY GREEN.

- Scheduled_Time_Of_Change AND Street DEFINE THE STATE OF THE TRAFFIC LIGHT. Street IS THE ONE THAT HAS THE GREEN LIGHT AND Scheduled_Time_Of_Change IS THE TIME OF THE NEXT SCHEDULED LIGHT CHANGE. (THIS IS DETERMINED BY Change_State, WHICH WE WILL LOOK AT SOON.)

- Time_Of_Last_Change KEEPS TRACK OF WHEN THE LIGHT WAS LAST CHANGED.

Traffic_Control_Package - Body

```
with Traffic_Package, Traffic_Flow_Package;
package body Traffic_Control_Package;

   use Calendar;

   Min_Change_Time : constant Duration := 30.0;
   Max_Change_Time : constant Duration := 3.0 * 60;

   Time_Of_Last_Change      : Time := Clock;
   Scheduled_Time_Of_Change : Time := Time_Of_Last_Change + Max_Change_Time / 2;

   Min_Expired : Boolean := False;

   type Street_Type is (Street_One, Street_Two);
   Street : Street_Type := Street_One;

   procedure Change_State
      (Street                   : in out Street_Type;
       Scheduled_Time_Of_Change : in out Time;
       Time_Of_Last_Change      : in out Time;
       Min_Expired              : out Boolean) is separate;

   task body Traffic_Control_Task is separate;

end Traffic_Control_Package;
```

- AFTER STARTING THE TRAFFIC LIGHT, THE TASK ENTERS THE LOOP FOREVER. (NOTE THAT SINCE WE DO NOT EXPECT A TRAFFIC LIGHT TO TERMINATE GRACEFULLY - THEY ONLY TERMINATE WHEN DESTROYED BY CARS OR TRUCKS - WE DO NOT NEED A TERMINATE ALTERNATIVE.)

- THE LOOP CONSISTS OF THE SINGLE SELECT STATEMENT. AFTER THE LIGHT IS FIRST STARTED UP, OR AFTER A LIGHT CHANGE, ONLY CALLS TO CHANGE TO THE EMERGENCY LIGHT PATTERN SHOULD BE ALLOWED. THE Crossing_Button IS ONLY ACCEPTED WHEN ITS GUARD IT TRUE. THIS IS SET BY THE SECOND DELAY ALTERNATIVE AFTER 30 SECONDS HAVE PASSED. IT IS RESET WHENEVER THE STATE OF THE TRAFFIC LIGHT IS CHANGED.

- WHEN THE URGENT ENTRY CALL TO Start_Emergency_Light_Pattern IS ISSUED, THE FIRST ACCEPT ALTERNATIVE OF THE SELECT STATEMENT IS SELECTED. THIS CAUSES THE EMERGENCY LIGHT PATTERN TO BE SET. THE TASK THEN WAITS AT THE ACCEPT STATEMENT FOR End_Emergency_Light_Pattern. WHEN THIS LAST ENTRY IS CALLED, THE STATE OF THE TRAFFIC LIGHT IS CHANGED AND THE TRAFFIC LIGHT IS STARTED UP AGAIN.

- NOTE THAT Start_Emergency_Light_Pattern ESSENTIALLY ASKS THIS TASK TO PERFORM A SERVICE AND THEN SUSPEND ITSELF, WHILE End_Emergency_Light_Pattern IS A REQUEST FOR THIS TASK TO REACTIVATE ITSELF.

- ASK THE STUDENTS WHAT THEY THINK WILL HAPPEN IF THE GUARD ON THE SECOND DELAY ALTERNATIVE IS REMOVED. ASK THEM IF THEY THINK THE PROGRAM WILL "WORK" - IT WILL, ASSUMING A FAIR RUNTIME SYSTEM. ASK THEM WHAT THEY THINK THE EFFECT ON PERFORMANCE WILL BE - ONCE THE DELAY IS ZERO OR NEGATIVE, THE TASK WILL WASTE TIME ASSIGNING True TO Min_Expired EACH TIME THROUGH THE LOOP, UNLESS AN ENTRY CALL TO ONE OF THE ENTRIES IS WAITING FOR RENDEZVOUS, OR THE OTHER DELAY EXPIRES. (IT IS WHEN THE OTHER DELAY EXPIRES THAT WE NEED A FAIR RUNTIME SYSTEM TO ENSURE THAT EVENTUALLY THAT ALTERNATIVE WILL BE SELECTED.)

Traffic_Control_Package - Traffic_Control_Task Body

```
separate (Traffic_Control_Package)
task body Traffic_Control_Task is
   Street_One_Traffic_Flow : Traffic_Flow_Package.Traffic_Flow_Task_Type;
   Street_Two_Traffic_Flow : Traffic_Flow_Package.Traffic_Flow_Task_Type;
begin -- Traffic_Control_Task
   Traffic_Package.Start_Light;
   loop
      select
         accept Start_Emergency_Light_Pattern;
             Traffic_Package.Set_Emergency_Light_Pattern;
         accept End_Emergency_Light_Pattern;
         Change_State
             (Street, Scheduled_Time_Of_Change, Time_Of_Last_Change, Min_Expired);
         Traffic_Package.Start_Light;
      or
         delay Scheduled_Time_Of_Change - Clock;
         Change_State
             (Street, Scheduled_Time_Of_Change, Time_Of_Last_Change, Min_Expired);
         Traffic_Package.Change_Light;
      or
         when not Min_Expired =>
             delay (Time_Of_Last_Change + Min_Change_Time) - Clock;
             Min_Expired := True;
      or
         when Min_Expired =>
             accept Crossing_Button;
             Change_State
                (Street, Scheduled_Time_Of_Change, Time_Of_Last_Change, Min_Expired);
             Traffic_Package.Change_Light;
      end select;
   end loop;
end Traffic_Control_Task;
```

INSTRUCTOR NOTES

- THIS PROCEDURE SIMPLY CHANGES THE STATE OF THE TRAFFIC LIGHT.

- WHICHEVER STREET LAST HAD THE GREEN LIGHT NOW GETS THE RED LIGHT. IF NEITHER STREET HAS HAD ANY TRAFFIC IN THE LAST 10 MINUTES, THEN ONE-HALF OF THE MAXIMUM TIME CHANGE IS USED AS THE ELAPSED TIME UNTIL ANOTHER TIME CHANGE MUST OCCUR. OTHERWISE, A WEIGHTED AVERAGE IS USED, BASED ON THE TRAFFIC FLOW IN EACH DIRECTION.

- WHEN PRESENTING THIS SLIDE, JUST EMPHASIZE THE STATE CHANGE, AND THE USE OF THE Obtain_Traffic_Flow ENTRIES. DO NOT GO INTO THE DETAILS OF THE CALCULATIONS.

Traffic_Control_Package - Change_State

```
separate (Traffic_Control_Package)
procedure Change_State
  (Current_Street           : in out Street_Type;
   Scheduled_Time_Of_Change : in out Time;
   Time_Of_Last_Change      : in out Time;
   Min_Expired              : out Boolean) is
   Street_Count : array (Street_Type) of Natural;
   Total_Count  : Natural;
   Time_Interval             : constant Duration :=
                                 Max_Change_Time - Min_Change_Time;
   Elapsed_Time_Until_Change : Duration;
begin
   Min_Expired := False;
   if Current_Street = Street_One then
      Current_Street := Street_Two;
   else
      Current_Street := Street_One;
   end if;
   Street_One_Traffic_Flow.Obtain_Traffic_Flow (Street_Count (Street_One));
   Street_Two_Traffic_Flow.Obtain_Traffic_Flow (Street_Count (Street_Two));
   Total_Count := Street_Count (Street_One) + Street_Count (Street_Two);
   if Total_Count = 0 then
      Elapsed_Time_Until_Change := Max_Change_Time / 2;
   else
      Elapsed_Time_Until_Change := Min_Change_Time +
             Time_Interval * Street_Count (Current_Street) / Total_Count;
   end if;
   Time_Of_Last_Change := Clock;
   Schedule_Time_Of_Change := Time_Of_Last_Change + Elapsed_Time_Until_Change;

end Change_State;
```

INSTRUCTOR NOTES

- ALLOW 30 MINUTES FOR THIS SECTION.

- THIS SECTION DESCRIBES THE TIME ENTRY CALL AND THE CONDITIONAL ENTRY CALL.

- THE MAIN MESSAGES ARE THAT A PROGRAM CAN SPECIFY ACTIONS TO BE TAKEN.
  - IF AN ENTRY CALL CANNOT BE ACCEPTED IMMEDIATELY, OR
  - IF AN ENTRY CALL CANNOT BE ACCEPTED WITHIN A SPECIFIED AMOUNT OF TIME.

- THIS SECTION ALSO DISCUSSES THE Count ATTRIBUTE.

Section 9

SELECT STATEMENTS FOR MAKING ENTRY CALLS

INSTRUCTOR NOTES

- THIS SLIDE IS MOTIVATION FOR WHY WE MIGHT NEED MORE THAN A SIMPLE ENTRY CALL.

- BULLET 2 - THE TWO ITEMS ARE THE SAME ONES USED IN THE BEGINNING OF SECTION 2.

TIMED AND CONDITIONAL ENTRY CALLS - OVERVIEW

- IN THE ENTRY CALL / accept STATEMENT MODEL WE HAVE SEEN SO FAR, WHEN A TASK MAKES AN ENTRY CALL IT WAITS UNTIL THE RENDEZVOUS IS COMPLETE.

- FOR SOME TASKS, THIS IS INADEQUATE. SOMETIMES TASKS NEED MORE CONTROL OVER ENTRY CALLS THAT THEY MAKE.
  - IF A TASK'S ENTRY CALL IS NOT ACCEPTED WITHIN SOME SPECIFIED TIME, THEN THE TASK MAY WISH TO PERFORM SOME OTHER ACTION.
  - IF A TASK'S ENTRY CALL CANNOT BE ACCEPTED <u>IMMEDIATELY</u>, THEN THE TASK MAY WISH TO PERFORM SOME OTHER ACTION.

- Ada PROVIDES FOR THESE CASES WITH SELECT STATEMENTS OF THE FORM:
  - timed entry call
  - conditional entry call

INSTRUCTOR NOTES

- THIS SLIDE MOTIVATES THE NEED FOR A TIMED ENTRY CALL.

PROBLEM 1 - WHEN AN ENTRY CALL IS NOT ACCEPTED IN TIME

- IF A TASK'S ENTRY CALL IS NOT ACCEPTED WITHIN SOME SPECIFIED TIME, THEN THE TASK MAY WISH TO PERFORM SOME OTHER ACTION.

- SUPPOSE THE TEMPERATURE OF A WATER-COOLED DEVICE IS TO BE MAINTAINED BELOW A CERTAIN TEMPERATURE. IF THE TEMPERATURE RISES ABOVE THAT LEVEL THEN THE WATER FLOW IS TO BE INCREASED UNTIL THE TEMPERATURE FALLS BY 10 DEGREES. A TASK TO PERFORM THESE ACTIONS MIGHT CONTAIN THE FOLLOWING CODE FRAGMENT:

```
loop
   Temperature_Task.Read (Temperature => Current_Temperature);

   if Current_Temperature >= Standard_Temperature then

      Water_Control_Task.Increase_Water_Flow;

      while Current_Temperature >= Standard_Temperature - 10 loop
         Temperature_Task.Read (Temperature => Current_Temperature);
      end loop;

      Water_Control_Task.Normal_Water_Flow;

   end if;

end loop;
```

- WHAT HAPPENS IF THE SENSORS THAT MONITOR THE TEMPERATURE BREAK, AND PREVENT THE Temperature_Task FROM RESPONDING?

- WHAT WE NEED IS A WAY TO TAKE SOME CORRECTIVE ACTION IF THE Read ENTRY OF THE Temperature_Task IS NOT ACCEPTED WITHIN SOME REASONABLE TIME.

INSTRUCTOR NOTES

- NOTE THE NESTING OF A TIMED ENTRY CALL WITHIN A TIMED ENTRY CALL.
- TO SIMPLIFY THE EXAMPLE, WE ASSUME THAT WE HAVE NO PROBLEMS WITH THE VALVE.

A SOLUTION USING TIMED ENTRY CALLS

```
loop
   select
      Temperature_Task.Read (Temperature => Current_Temperature);
      if Current_Temperature> = Standard_Temperature then
         Water_Control_Task.Increase_Water_Flow;
         while Current_Temperature> = Standard_Temperature - 10 loop
            select
               Temperature_Task.Read (Temperature => Current_Temperature);
            or
               delay 0.1;
               raise Temperature_Exception;
            end select;
         end loop;
         Water_Control_Task.Normal_Water_Flow;
      end if;
   or
      delay 0.1;
      raise Temperature_Exception;
   end select;
end loop;
```

INSTRUCTOR NOTES

- MAKE SURE THE CLASS UNDERSTANDS THAT WHILE A SELECTIVE WAIT CAN CONTAIN MORE THAN ONE ACCEPT STATEMENT, A TIMED ENTRY CALL IS FOR ONE ENTRY CALL.

- MAKE SURE THE CLASS UNDERSTANDS THAT THE CALL IS ISSUED, EVEN IF IT IS LATER CANCELLED. THIS WILL BE NEEDED WHEN WE TALK ABOUT THE COUNT ATTRIBUTE.

TIMED ENTRY CALLS

- TIMED ENTRY CALLS ALLOW A TASK TO PERFORM SOME OTHER ACTIONS IF ITS ENTRY CALL IS NOT ACCEPTED IN TIME.

```
select
    entry call statement ;
  [ sequence of statements ]
or
    delay  delay expression ;
  [ sequence of statements ]
end select;
```

- EXECUTION OF THE TIMED ENTRY CALL PROCEEDS AS FOLLOWS:
  - THE ACTUAL PARAMETERS, IF ANY, ARE EVALUATED.
  - THE DELAY EXPRESSION IS EVALUATED AND THEN THE ENTRY CALL IS ISSUED.
    - IF THE RENDEZVOUS CAN BE STARTED WITHIN THE SPECIFIED DELAY, THEN IT IS PERFORMED AND THE FIRST SEQUENCE OF STATEMENTS, IF ANY, IS EXECUTED.
    - OTHERWISE, THE ENTRY CALL IS CANCELLED AFTER THE SPECIFIED DELAY HAS EXPIRED, AFTER WHICH THE SECOND SEQUENCE OF STATEMENTS IS EXECUTED.
    - CANCELLING THE ENTRY CALL REMOVES THE CALL FROM THE ENTRY'S QUEUE. THE CALLED TASK NEVER SEES THE CALL.

INSTRUCTOR NOTES

- Fast_Disk IS THE DISK MANAGER TASK FOR A FAST DISK. Allocation_Request IS THE SIZE OF THE ALLOCATION NEEDED. IT COULD BE IN SECTORS, PAGES, ETC. THE ADDRESS WHERE THE SPACE IS ALLOCATED IS RETURNED IN Address_Allocated. THIS COULD BE A SECTOR ADDRESS, PAGE ADDRESS, ETC.

PROBLEM 2 - WHEN RENDEZVOUS CANNOT OCCUR IMMEDIATELY

- IF A TASK'S ENTRY CALL CANNOT BE ACCEPTED <u>IMMEDIATELY</u>, THEN THE TASK MAY WISH TO PERFORM SOME OTHER ACTIONS.

- CONSIDER THE FOLLOWING PROBLEM. A TASK WANTS TO ALLOCATE SPACE ON A FAST DISK. IT MIGHT CONTAIN A CODE FRAGMENT SUCH AS:

```
Fast_Disk.Allocate (Allocate_Requested => Size_Needed,
                    Address_Allocated  => Start_Address);
```

- IF THE FASTER DISK IS NOT READY TO ACCEPT ALLOCATION REQUESTS, THE CALLING TASK WAITS.

- SUPPOSE THERE IS A SLOWER DISK AVAILABLE THAT THE CALLING TASK WOULD BE WILLING TO USE IF THE FASTER ONE IS NOT IMMEDIATELY AVAILABLE.

- WE WANT A WAY TO ASK FOR THE SLOWER DISK IF THE FASTER ONE IS NOT IMMEDIATELY AVAILABLE. IF NEITHER DISK IS AVAILABLE IMMEDIATELY, THEN WE WANT TO PERFORM SOME OTHER ACTION, SUCH AS DELAYING FOR A SECOND BEFORE REQUESTING THE ALLOCATION AGAIN.

INSTRUCTOR NOTES

- IF THE FASTER DISK IS NOT AVAILABLE IMMEDIATELY, THEN THE TASK ATTEMPTS TO ALLOCATE SPACE ON THE SLOWER DISK. IF THE SLOWER DISK IS NOT AVAILABLE, THEN WE WAIT FOR A SECOND AND TRY ALL OVER AGAIN.

- TO AVOID BUSY WAITING, WE DELAY FOR ONE SECOND. THIS FREES THE PROCESSOR TO POSSIBLY DO SOME OTHER USEFUL WORK.

A SOLUTION USING CONDITIONAL ENTRY CALLS

```
Allocated := False;
loop

   select

      Fast_Disk.Allocate (Allocation_Requested => Size_Needed,
                          Address_Allocated    => Start_Address);
      Allocated := True;

   else

      select

         Slow_Disk.Allocate (Allocation_Requested => Size Needed,
                             Address_Allocated    => Start_Address);
         Allocated := True;

      else

         null;

      end select;

   end select;

   exit when Allocated;
   delay 1.0;

end loop;
```

INSTRUCTOR NOTES

- THE RENDEZVOUS MAY FAIL TO OCCUR IMMEDIATELY BECAUSE

  1. THERE ARE OTHER CALLS QUEUED FOR THE ENTRY, OR

  2. THE CALLED TASK IS NOT WAITING AT AN ACCEPT STATEMENT FOR THE ENTRY, OR

  3. THE CALLED TASK IS NOT WAITING AT A SELECTIVE WAIT STATEMENT HAVING AN OPEN ACCEPT ALTERNATIVE FOR THE ENTRY.

- AGAIN, MAKE SURE THAT THE STUDENTS UNDERSTAND THAT A CONDITIONAL ENTRY CALL IS FOR ONE ENTRY CALL.

CONDITIONAL ENTRY CALL

- CONDITIONAL ENTRY CALLS ALLOW A TASK TO PERFORM SOME OTHER ACTION IF THE ENTRY CALL CANNOT RESULT IN AN IMMEDIATE RENDEZVOUS.

```
select
   entry call statement ;
[  sequence of statements  ]
else
   sequence of statements
end select;
```

- EXECUTION OF A CONDITIONAL ENTRY CALL PROCEEDS AS FOLLOWS:
  - THE ACTUAL PARAMETERS, IF ANY, ARE EVALUATED.
  - THE ENTRY CALL IS ISSUED.
    - IF THE CALLED TASK CAN ESTABLISH A RENDEZVOUS WITH THE CALLING TASK IMMEDIATELY THEN THE RENDEZVOUS TAKES PLACE, AND THE FIRST SEQUENCE OF STATEMENTS IS EXECUTED.
    - OTHERWISE, IF THE RENDEZVOUS CANNOT TAKE PLACE IMMEDIATELY THEN THE ENTRY CALL IS <u>CANCELLED</u> AND THE SECOND SEQUENCE OF STATEMENTS IS EXECUTED.

INSTRUCTOR NOTES

- THIS SLIDE DESCRIBES THE Count ATTRIBUTE AND THE EFFECT OF TIMED AND CONDITIONAL ENTRY CALLS ON THE QUEUE.

- BULLET 2, ITEM 1 - IF Entry_A'Count IS EVALUATED WITHIN AN ACCEPT STATEMENT FOR Entry_A, THEN Entry_A'Count DOES NOT INCLUDE THE CALLING TASK. (THE CALL HAS ALREADY BEEN REMOVED FROM THE QUEUE.)

- BULLET 2, ITEM 2 - EVALUATION OF THE COUNT ATTRIBUTE IS ALSO EXCLUDED FROM PACKAGE BODIES AND OTHER TASKS WITHIN THE TASK.

'Count ATTRIBUTE

- EACH TASK ENTRY HAS A QUEUE ASSOCIATED WITH IT.
  - WHEN A TASK ISSUES AN ENTRY CALL, THE TASK IS PLACED ON THAT ENTRY'S QUEUE.
  - WHEN THE RENDEZVOUS STARTS, THE TASK IS REMOVED FROM THE QUEUE.

- Ada PROVIDES A WAY FOR A TASK TO DETERMINE HOW MANY TASKS ARE QUEUED ON ONE OF ITS ENTRIES. THIS IS PROVIDED BY THE Count ATTRIBUTE.
  - Entry_A'Count IS THE NUMBER OF ENTRY CALLS QUEUED ON Entry_A.
    - A TIMED ENTRY CALL TO Entry_A INCREASES Entry_A'Count BY 1 WHEN THE CALL IS ISSUED AND DECREASES IT BY ONE IF THE ENTRY CALL IS CANCELLED, SO CARE MUST BE TAKEN IN USING THIS ATTRIBUTE WHEN SUCH ENTRY CALLS ARE ALLOWED.
  - AS WITH ACCEPT STATEMENTS, THE Count ATTRIBUTE FOR A TASK'S ENTRY MAY ONLY BE USED WITHIN THE TASK. MOREOVER, IT MAY <u>NOT</u> BE USED WITHIN A SUBPROGRAM APPEARING WITHIN A TASK.

INSTRUCTOR NOTES

- THIS SLIDE SUGGESTS TWO POSSIBLE USES FOR THE Count ATTRIBUTE.

- IN THE FIRST EXAMPLE, ONCE WE ACCEPT A Button CALL, WE ENTER A LOOP TO CLEAR OUT THE Button QUEUE. SINCE NO CONDITIONAL OR TIMED ENTRY CALLS CAN BE ISSUED WHEN Button'Count EVALUATES TO NON-ZERO, WE KNOW THAT THERE IS AT LEAST ONE TASK WAITING FOR RENDEZVOUS.

- IN THE SECOND EXAMPLE, CONDITIONAL AND TIMED ENTRY CALLS CAN OCCUR. WITHOUT THE SELECT STATEMENT WE COULD HAVE A PROBLEM. SUPPOSE WE HAD WRITTEN THE LOOP WITHOUT THE SELECT STATEMENT AS:

```
for I in 1 .. Read_Request'Count
   accept Read_Request;
   Number_Of_Tasks_Reading := Number_Of_Tasks_Reading + 1;
end loop;
```

  MOREOVER, SUPPOSE THAT TASKS A AND B ARE THE ONLY TASKS CALLING THIS DATABASE TASK, AND THAT A AND B ARE WAITING ON THE Read_Request QUEUE. THE ACCEPT STATEMENT OUTSIDE OF THE LOOP WILL RESULT IN A RENDEZVOUS WITH TASK A AND WILL CAUSE TASK A TO BE REMOVED FROM THE QUEUE. NEXT, Read_Request'Count IS EVALUATED. SINCE B IS STILL ON THE QUEUE, THE LOOP WILL BE EXECUTED ONCE. NOW SUPPOSE THAT TASK B HAS ISSUED A TIMED ENTRY CALL TO Read_Request THAT NOW TIMES OUT. NOW THERE ARE NO TASKS WAITING TO RENDEZVOUS WITH THE DATABASE TASK, SO IT WILL SIT AT THE ACCEPT STATEMENT FOR Read_Request UNTIL A CALL TO Read_Request IS ISSUED.

- COMPARE THE USE OF THE for I FORM OF LOOP WITH

```
while Read_Request'Count > 0 loop
```

  THIS FORM EVALUATES THE COUNT ATTRIBUTE REPEATEDLY. IF NEW CALLS ARRIVE, THE LOOP WILL BE EXECUTED FOR EACH NEW CALL. THE for Loop FORM ENSURES THAT JUST THE CALLS WAITING WHEN, SAY, THE WRITE OPERATION FINISHES, ARE ACCEPTED.

'Count ATTRIBUTE - EXAMPLES

- THE Count ATTRIBUTE CAN BE USED TO GET RID OF EXTRANEOUS ENTRY CALLS:

  - A TRAFFIC LIGHT MAY HAVE A CROSSING BUTTON THAT CAN BE PUSHED SEVERAL TIMES BY AN IMPATIENT PEDESTRIAN. ASSUMING A TRAFFIC LIGHT TASK HAS AN ENTRY NAMED Button TO HANDLE THE BUTTON, IT CAN USE THE Count ATTRIBUTE TO CLEAR OUT THE BUTTON QUEUE. (ASSUME TIMED CALLS DO NOT OCCUR.)

```
accept Button do
   while Button'Count > 0 loop
      accept Button;
   end loop;
   ...
end Button;
```

- THE Count ATTRIBUTE CAN BE USED TO ACCEPT A CALL FROM ALL TASKS WAITING ON A QUEUE TO PROCEED.

  - A DATABASE TASK MAY PROTECT AGAINST SIMULTANEOUS READING AND WRITING, BUT IT MAY ALLOW MULTIPLE READERS TO PROCEED CONCURRENTLY. WHEN READING IS ALLOWED, THE Count ATTRIBUTE CAN BE USED TO ALLOW WAITING TASKS TO BEGIN READING.

```
accept Read_Request do
   Number_Of_Tasks_Reading := 1;
   for I in 1 .. Read_Request'Count loop

         select
            accept Read_Request;
            Number_Of_Tasks_Reading := Number_Of_Tasks_Reading + 1;
         else
            exit;  -- reached only if call cancelled after Count evaluated
         end select;

   end loop;
end Read_Request;
```

INSTRUCTOR NOTES

- ALLOW 60 MINUTES FOR THIS SECTION.

- RECOMMEND THAT STUDENTS READ EXERCISE 3.2 IN THE REAL-TIME ADA WORKBOOK.

Section 10

AVOIDING DEADLOCK

INSTRUCTOR NOTES

DEADLOCK

- DEADLOCK OCCURS WHEN TWO OR MORE TASKS ARE WAITING FOR CONDITIONS WHICH WILL NEVER HOLD.
  - EACH TASK IS WAITING FOR A CONDITION WHICH CAN ONLY BE SATISFIED BY ONE OF THE OTHER TASKS.
  - SINCE EACH TASK EXPECTS ONE OF THE OTHER TASKS TO RESOLVE THE CONFLICT, NONE OF THE TASKS CAN CONTINUE.
- THIS SECTION PRESENTS GUIDELINES FOR ENSURING THAT DEADLOCK DOES NOT OCCUR.
  - IT IS POSSIBLE TO WRITE CORRECT, DEADLOCK-FREE PROGRAMS THAT VIOLATE THESE GUIDELINES.
  - IF THESE GUIDELINES ARE FOLLOWED, IT IS EASIER TO ESTABLISH THAT DEADLOCK CANNOT OCCUR.

INSTRUCTOR NOTES

- BULLET 4 - THE NEXT SLIDE EXPANDS ON THIS.

GUIDELINES FOR AVOIDING DEADLOCK

- DESIGN TASKS SO THAT ENTRY CALLS GO IN ONE DIRECTION.
  - HIERARCHICAL DESIGN
  - NO TASK SHOULD BE A USER AND A SERVER TO THE SAME TASK

- A SEQUENCE OF ENTRY CALLS IS <u>SAFE</u> IF FOR EACH ENTRY IN THE SEQUENCE, AN ACCEPT STATEMENT IS REACHED FOR THE ENTRY WHENEVER ACCEPT STATEMENTS ARE REACHED FOR THE PREVIOUS ENTRIES IN THE SEQUENCE.

- DEADLOCK CAN BE AVOIDED BY ENSURING THAT ALL TASKS EXECUTE SAFE SEQUENCES OF ENTRY CALLS.

INSTRUCTOR NOTES

- IN TRYING TO DETERMINE WHETHER A SEQUENCE IS SAFE WE NEED TO ENSURE THAT TASKS DECLARED WITHIN NESTED MASTERS EVENTUALLY TERMINATE. ENSURING THIS SEPARATELY SIMPLIFIES THE VERIFICATION OF SAFENESS.

- IN THE EXAMPLE, AN ENTRY CALL TO A.Some_Entry IS SAFE BUT THE BLOCK IS NEVER LEFT BECAUSE IT REQUIRES TERMINATION OF TASK B. HOWEVER, B CANNOT TERMINATE UNTIL THE RENDEZVOUS COMPLETES, BUT THE RENDEZVOUS CANNOT OCCUR UNTIL THE BLOCK IS LEFT.

DEADLOCK AND DEPENDING ON MASTERS

- WHEN DETERMINING WHETHER A SEQUENCE OF ENTRY CALLS IS SAFE
  - TREAT DEPENDENCE ON NESTED MASTERS SEPARATELY
  - ENSURE THAT TASKS DEPENDING ON NESTED MASTERS WILL TERMINATE SO THAT THE MASTER CAN BE LEFT

```
task A is
   entry Some_Entry;
end A;

task body A is
   ...
   declare
      ...
      task B is ...
      task body B is
         ...
         A.Some_Entry;
         ...
      end B;
   begin -- block
      ...
   end;  -- block waits for B to terminate
   ...
   accept Some_Entry;  -- never reached
   ...
end A;
```

- AN ENTRY CALL TO A.Some_Entry <u>IS</u> SAFE, BUT DEADLOCK STILL OCCURS.

INSTRUCTOR NOTES

- THIS EXAMPLE IS BASED ON AN EXAMPLE IN Ada Software Design Methods Formulation Case Studies SofTech, August 1982, PREPARED UNDER CONTRACT DAAK80-80-C-0187.

- WE WILL SOLVE THIS PROBLEM AND SEE HOW DEADLOCK CAN SNEAK IN. WE WILL ALSO SEE THAT AVOIDING DEADLOCK MAY CAUSE OTHER PROBLEMS, IF WE ARE NOT CAREFUL.

- THIS IS THE ONLY EXAMPLE WE GIVE OF EXPLICIT TASK COMMUNICATION.

AVOIDING DEADLOCK IN A RADAR SYSTEM

- A RADAR SWEEPS OVER N SECTORS IN A SINGLE SCAN. AFTER EACH SECTOR IS SCANNED, THE BLIPS DETECTED IN IT MUST BE PROCESSED BEFORE THIS SECTOR IS SWEPT BY THE NEXT RADAR SCAN. EACH SECTOR UNDERGOES IDENTICAL PROCESSING AND EACH SECTOR CAN BE PROCESSED ASYNCHRONOUSLY.

- WE MODEL THE PROCESSING FOR EACH SECTOR AS A SINGLE TASK.

- WE REFORMULATE THE PROBLEM AS FOLLOWS: THE PROCESSING FOR A GIVEN SECTOR MUST BE COMPLETED IN THE TIME THE RADAR TAKES TO SWEEP N-1 SECTORS. IF PROCESSING CANNOT BE COMPLETED WITHIN THIS TIME, IT MUST BE STOPPED (PREEMPTED) AND RESTARTED WITH THE FRESH DATA FROM THE NEW SWEEP OF THE SECTOR.

- THE SOLUTION HAS THE FORM:

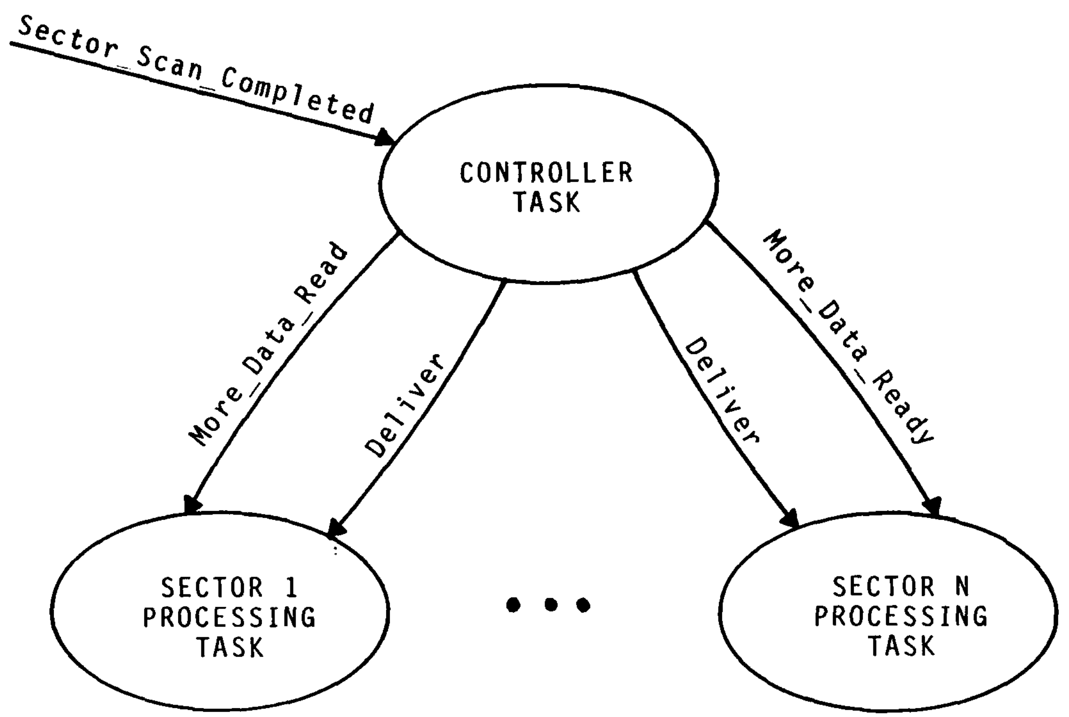
```
loop
   [GET NEW DATA FOR SECTOR];
   while [MORE BLIPS TO PROCESS] and not [PREEMPTED] loop

      [PROCESS A BLIP];
      [CHECK FOR PREEMPTION];

   end loop;

end loop;
```

INSTRUCTOR NOTES

- THIS SLIDE SHOWS THE TASK HIERARCHY.

- THE DESCRIPTIONS OF THE TASKS ARE GIVEN IN THE INSTRUCTOR NOTES FOR THE NEXT SLIDE.

AVOIDING DEADLOCK IN A RADAR SYSTEM - TASK HIERARCHY



INSTRUCTOR NOTES

- DO NOT SPEND TOO MUCH TIME ON THIS SLIDE.

- AS EACH SECTOR IS SCANNED, THE Sector_Processing_Controller_Task IS NOTIFIED VIA THE Sector_Scan_Completed ENTRY. THE Sector_Processing_Controller_Task ACCEPTS THIS ENTRY PRIOR TO STARTING UP PROCESSING OF A SECTOR'S BLIPS. THUS THE ENTRY IS USED TO SYNCHRONIZE WITH THE RADAR.

- PROCESSING FOR EACH SECTOR IS PERFORMED BY A TASK OF TYPE Sector_Processing_Task_Type. NEW SECTOR DATA IS DELIVERED TO THIS TASK VIA THE Deliver ENTRY. IN ORDER TO ENSURE THAT PROCESSING OF A SECTOR'S BLIPS FINISHES AFTER THE PREVIOUS N - 1 SECTORS HAVE BEEN PROCESSED, THE More_Data_Ready ENTRY IS CALLED.

- POINT OUT THAT WE HAVE AN ARRAY OF TASKS OF TYPE Sector_Processing_Task_Type, ONE FOR EACH SECTOR.

AVOIDING DEADLOCK IN A RADAR SYSTEM - CONTINUED

```
task Sector_Processing_Controller_Task is

   entry Sector_Scan_Completed (New_Sector_Data : in Sector_Data_Type);

end Sector_Processing_Controller_Task;

task body Sector_Processing_Controller_Task is

   Current_Sector : Positive Range 1 .. Number_Of_Sectors := 1;

   task type Sector_Processor_Task_Type is

      entry Deliver (New_Sector_Data : in Sector_Data_Type);
      entry More_Data_Ready;

   end Sector_Processor_Task_Type;

   Sector_Processor_List : array (1 .. Number_Of_Sectors)
                              of Sector_Processor_Task_Type;

   task body Sector_Processor_Task_Type is separate;

begin

 STATEMENTS ON NEXT SLIDE

end Sector_Processor_Task_Type;
```

INSTRUCTOR NOTES

- THE TASK DELIVERS NEW SECTOR DATA (BLIPS) TO THE SECTOR PROCESSOR TASK. IT DOES THIS AFTER SYNCHRONIZING WITH THE RADAR.

- TO ENSURE THAT THE SECTOR PROCESSOR TASK WILL BE READY, THE CONTROLLER TASK PREEMPTS PROCESSING OF THE NEXT SECTOR BY CALLING More_Data_Ready.

AVOIDING DEADLOCK IN A RADAR SYSTEM - AN INITIAL ATTEMPT

```
task body Sector_Processing_Controller_Task is

 DECLARATIONS ON PREVIOUS SLIDE

begin

   for I in 1 .. Number_Of_Sectors loop
      accept Sector_Scan_Completed (New_Sector_Data : in Sector_Data_Type) do
         Sector_Data := New_Sector_Data;
      end Sector_Scan_Completed;
      Sector_Processor_List (I).Deliver (Sector_Data);
   end loop;

   loop

      Sector_Processor_List (Current_Sector).More_Data_Ready;

      accept Sector_Scan_Completed (New_Sector_Data : in Sector_Data_Type) do
         Sector_Data := New_Sector_Data;
      end Sector_Scan_Completed;
      Sector_Processor_List (Current_Sector).Deliver (Sector_Data);
      if Current_Sector = Number_Of_Sectors then
         Current_Sector := 1;
      else
         Current_Sector := Current_Sector + 1;
      end if;

   end loop;

end Sector_Processing_Controller_Task;
```

INSTRUCTOR NOTES

- ONCE DATA FOR THE SECTOR IS DELIVERED, IT IS PROCESSED BLIP AT A TIME. AS PROCESSING OF EACH BLIP IS FINISHED, A More_Data_Ready ENTRY CALL CAN BE ACCEPTED. IF A CALL TO More_Data_Read HAS BEEN ISSUED, THEN PROCESSING OF THE REMAINING BLIPS IS ABANDONED.

- THERE IS A PROBLEM WITH THIS SOLUTION. UNLESS PRESSED FOR TIME, ASK THE CLASS IF THEY KNOW WHAT IT IS.

- THE PROBLEM OCCURS IF THE PROCESSING OF BLIPS FINISHES BEFORE THE More_Data_Ready CALL IS ISSUED. IN THIS CASE, THE TASK FOR THIS SECTOR WILL WAIT FOR A CALL TO Deliver WHILE THE CONTROLLER TASK WAITS FOR A RENDEZVOUS WITH THE More_Data_Ready ENTRY. DEADLOCK OCCURS.

- THE NEXT SLIDE GOES INTO MORE DETAIL ON THE PROBLEM.

AVOIDING DEADLOCK IN A RADAR SYSTEM - AN INITIAL ATTEMPT (CONTINUED)

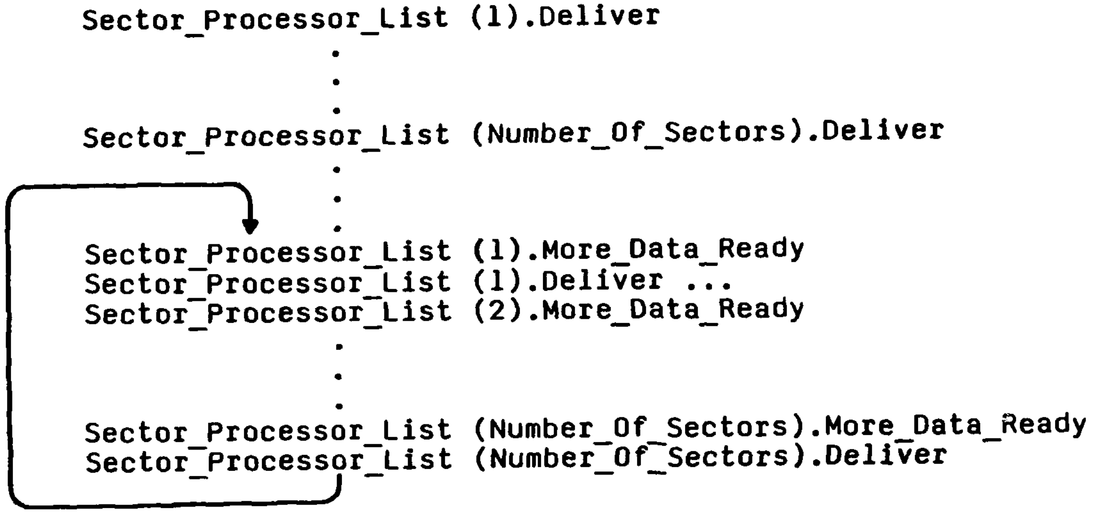
```
separate (Sector_Processing_Controller_Task)
task body Sector_Processor_Task_Type is

   Current_Sector_Data : Sector_Data_Type;

   procedure Process_Blip (Blip : in Blip_Type) is separate;  -- NOT SHOWN HERE

begin

   loop

      accept Delivery (New_Sector_Data : Sector_Data_Type) do
         Current_Sector_Data := New_Sector_Data;
      end Deliver;

      for Current_Blip in 1 .. Current_Sector_Data.Number_Of_Blips loop

         Process_Blip (Current_Sector_Data.Blip_List (Current_Blip));
         -- CHECK FOR PREEMPTION:
         select
            accept More_Data_Ready;
            exit;  -- for loop
         else
            null;
         end select;

      end loop;

   end loop;

end Sector_Processor_Task_Type;
```

INSTRUCTOR NOTES

- THE ARROW INDICATES A LOOP.

AVOIDING DEADLOCK IN A RADAR SYSTEM - UNSAFE SEQUENCE

- THE SEQUENCE OF ENTRY CALLS



```
Sector_Processor_List (1).Deliver
                   .
                   .
                   .
Sector_Processor_List (Number_Of_Sectors).Deliver
                   .
                   .
                   .
Sector_Processor_List (1).More_Data_Ready
Sector_Processor_List (1).Deliver ...
Sector_Processor_List (2).More_Data_Ready
                   .
                   .
                   .
Sector_Processor_List (Number_Of_Sectors).More_Data_Ready
Sector_Processor_List (Number_Of_Sectors).Deliver
```

IS NOT SAFE.

- FOR SOME SECTOR I, IF

```
Sector_Processor_List (I).More_Data_Ready
```

IS CALLED AFTER THE BLIPS OF SECTOR I HAVE BEEN PROCESSED, THE More_Data_Ready ENTRY WILL NOT BE ACCEPTED, THUS THE SEQUENCE IS NOT SAFE.

INSTRUCTOR NOTES

- TO FIX THE PREVIOUS PROBLEM, WE MODIFY THE TASK TO ACCEPT THE More_Data_Ready ENTRY IMMEDIATELY AFTER EACH BLIP IS PROCESSED. WHEN THE ENTRY IS ACCEPTED, THE Out_Of_Time EXCEPTION IS RAISED, WHICH FORCES US OUT OF THE LOOP. THE EXCEPTION HANDLER EXECUTES THE null STATEMENT, AFTER WHICH THE BLOCK IS LEFT. THE Deliver ENTRY IS NOW CALLED.

- THE ENTRY CALLS ON THIS TASK ARE SAFE

  - THE ACCEPT STATEMENTS DO NOT ENTER INTO ADDITIONAL RENDEZVOUS

  THEREFORE THE SEQUENCE OF ENTRY CALLS MADE BY THESE TASKS IS SAFE. SINCE THERE ARE NO DEPENDENT TASKS TO WORRY ABOUT, DEADLOCK DOES NOT OCCUR. (WE ARE ASSUMING THAT THE TASK CALLING Sector_Scan_Completed ALSO SATISFIES THESE REQUIRED CONDITIONS).

AVOIDING DEADLOCK IN A RADAR SYSTEM - A CORRECT SOLUTION

```
separate (Sector_Processing_Controller_Task)
task body Sector_Processor_Task_Type is

   Current_Sector_Data : Sector_Data_Type;

   procedure Process_Blip (Blip : in Blip_Type) is separate;

begin
   loop

      accept Deliver (New_Sector_Data :  Sector_Data_Type) do
         Current_Sector_Data := New_Sector_Data;
      end Deliver;

      declare
         Out_Of_Time : exception;
      begin

         for Current_Blip in 1 .. Current_Sector_Data.Number_Of_Blips loop
            Process_Blip (Current_Sector_Data.Blip_List (Current_Blip));

            select
               accept More_Data_Ready;
               raise Out_Of_Time;
            else
               null;
            end select;

         end loop;
         accept More_Data_Ready;
      exception
         when Out_Of_Time =>
            null;
      end; -- block

   end loop;
end Sector_Processor_Task_Type;
```

INSTRUCTOR NOTES

THE SEQUENCE OF ENTRY CALLS IS THE ONE DEPICTED ON 10-9.

AVOIDING DEADLOCK IN A RADAR SYSTEM

- THE SEQUENCE OF ENTRY CALLS ISSUED BY Sector_Processing_Controller_Task IS NOW SAFE.

- THE INITIAL CALLS TO Deliver EACH ARE ACCEPTED. IT IS THE FIRST ACTION TAKEN IN A SECTOR PROCESSING TASK.

- CONSIDER THE SEQUENCE

  ```
  Sector_Processor_List (I).More_Data_Ready
  Sector_Processor_List (I).Deliver
  ```

  WHEN THE More_Data_Ready ENTRY IS CALLED EITHER

  - THE BLIPS ARE BEING PROCESSED:
    - CALL IS ACCEPTED IN THE *for* loop
    - EXCEPTION IS RAISED
    - BLOCK IS LEFT, OR

  - PROCESSING OF BLIPS HAS FINISHED
    - CALL IS ACCEPTED OUTSIDE THE *for* loop
    - BLOCK IS LEFT

  IN EITHER CASE, THE ENTRY IS ACCEPTED AND THE BLOCK IS LEFT. THE Sector_Processor_List (I) TASK THEN EXECUTES AN ACCEPT STATEMENT FOR THE Deliver ENTRY.

INSTRUCTOR NOTES

- WE NOW TURN OUR ATTENTION TO RESOURCE SHARING.
- THIS IS JUST THE Buffer_Allocation_Type PRESENTED EARLIER IN THE COURSE.
- REVIEW THE USE OF THE TASK TYPE AND EXPLAIN WHAT THE GUARDS ARE DOING.
- DO NOT GO INTO DETAIL, JUST REVIEW THIS FOR THE NEXT FEW SLIDES.

BUFFER ALLOCATION REVISITED

```
subtype Buffer_Range_Type is range 1 .. Total_Buffers;

task type Buffer_Allocation_Type is
   entry Request (Buffer : out Buffer_Range_Type);
   entry Release (Buffer : in Buffer_Range_Type);
end Buffer_Allocation_Type;

task body Buffer_Allocation_Type is

   Available                   : array (Buffer_Range_Type) of Boolean :=
                                           (others => True);
   Number_Of_Buffers_Available : Natural := Total_Buffers;

begin -- Buffer_Allocation_Type
   loop
      select
         when Number_Of_Buffers_Available > 0 =>
            accept Request (Buffer : out Buffer_Range_Type) do
               Search_Loop:
                  for Candidate_Buffer in Buffer_Range_Type'Range loop
                     if Available (Candidate_Buffer) then
                        Available (Candidate_Buffer) := False;
                        Buffer := Candidate_Buffer;
                        exit Search_Loop;
                     end if;
                  end loop Search_Loop;
               Number_Of_Buffers_Available := Number_Of_Available_Buffers - 1;
            end Request;
      or
            accept Release (Buffer : in Buffer_Range_Type) do
               Available (Buffer) := True;
               Number_Of_Buffers_Available := Number_Of_Available_Buffers + 1;
            end Release;

      end select;
   end loop;
end Buffer_Allocation_Type;
```

INSTRUCTOR NOTES

- DO NOT EXPLAIN HOW TO SOLVE THIS DEADLOCK (THIS WILL BE DONE SHORTLY). JUST POINT OUT THAT THE DEADLOCK OCCURS BECAUSE TASK Here HAS ALREADY ACQUIRED TWO BUFFERS AND IS ATTEMPTING TO GET A THIRD ONE, WHILE TASK There HAS ALREADY ACQUIRED ONE BUFFER AND IS ATTEMPTING TO GET A SECOND ONE. NEITHER TASK CAN PROCEED UNTIL IT GETS A BUFFER HELD BY THE OTHER, AND NEITHER CAN GIVE UP ONE OF ITS BUFFERS UNTIL IT GETS ONE FROM THE OTHER.

DEADLOCK WHILE SHARING BUFFERS

- ASSUME THAT THERE ARE ONLY THREE BUFFERS (Total_Buffers = 3).

```
      Buffer_Allocation : Buffer_Allocation_Type;

                  ...

task body Here is                         task body There is
   ...                                       ...
begin                                     begin
   Buffer_Allocation.Request (H_1);               ...
      ...                                    Buffer_Allocation.Request (T_1);
   Buffer_Allocation.Request (H_2);               ...
      ...                                    Buffer_Allocation.Request (T_2); ⬅

➡ Buffer_Allocation.Request (H_3);
      ...                                         ...
   Buffer_Allocation.Release (H_3);

   Buffer_Allocation.Release (H_2);               ...
      ...                                    Buffer_Allocation.Release (T_2);
   Buffer_Allocation.Release (H_1);               ...
      ...                                    Buffer_Allocation.Release (T_1);

end Here;                                 end There;
```

- IF BOTH TASKS ARE AT THE ARROWS AT THE SAME TIME, THEY WILL BOTH BE BLOCKED AND WILL DEADLOCK.

INSTRUCTOR NOTES

- DO NOT EXPLAIN HOW TO SOLVE THIS DEADLOCK (THIS WILL BE EXPLAINED SHORTLY). JUST POINT OUT THAT THE DEADLOCK OCCURS BECAUSE TASK A HAS ACQUIRED EXCLUSIVE ACCESS TO File_1 AND IS ASKING FOR EXCLUSIVE ACCESS TO File_2, WHILE TASK B HAS ACQUIRED EXCLUSIVE ACCESS TO File_2 AND IS ASKING FOR EXCLUSIVE ACCESS TO File_1. BOTH TASKS ARE BLOCKED, WAITING FOR CONDITIONS THAT WILL NEVER HAPPEN.

DEADLOCK WHILE SHARING A FILE

- TWO TASKS, A AND B, BOTH NEED TO UPDATE TWO FILES. THEY DO SO BY CALLING A FILE MANAGER TASK THAT ACTS AS AN AGENT FOR TASKS SHARING THE FILE SYSTEM.

```
   task body A is                                task body B is
      ...                                           ...
   begin                                         begin
      ...                                           ...
      File_Manager.Open                             File_Manager.Open
         (File          => File_1,                     (File          => File_2,
          Access_Rights => Exclusive);                  Access_Rights => Exclusive);

➡     File_Manager.Open                             File_Manager.Open
         (File          => File_2,                     (File          => File_1,
          Access_Rights => Exclusive);                  Access_Rights => Exclusive);  ⬅
         ...                                           ...
      File_Manager.Close                            File_Manager.Close
         (File          => File_2,                     (File          => File_1,
          Access_Rights => Exclusive);                  Access_Rights => Exclusive);

      File_Manager.Close                            File_Manager.Close
         (File          => File_1,                     (File          => File_2,
          Access_Rights => Exclusive);                  Access_Rights => Exclusive);
         ...                                           ...
   end A;                                        end B;
```

- IF BOTH TASKS ARE EVER AT THE ARROWS AT THE SAME TIME, THEY WILL BOTH BE BLOCKED AND WILL DEADLOCK.

INSTRUCTOR NOTES

- BULLET 1 - THE TWO ITEMS EXPLAIN WHY DEADLOCK WILL NOT OCCUR IF THE TASKS HAD ONLY NEEDED ONE BUFFER EACH. CONTRAST THIS WITH THE PIECEMEAL ACQUISITION OF RESOURCES IN THE LAST TWO EXAMPLES.

WHAT WENT WRONG?

- IF THE TASKS SHARING THE BUFFERS ONLY NEEDED ONE BUFFER EACH. THEN THE TASKS COULD NOT DEADLOCK IN ATTEMPTING TO SHARE THIS RESOURCE.
  - A TASK ACQUIRES A BUFFER, USES IT, AND RETURNS IT TO THE BUFFER POOL WHEN FINISHED WITH IT.
  - IF A TASK DOES NOT FIND ANY BUFFERS AVAILABLE, IT WAITS UNTIL ANOTHER TASK FINISHES USING ONE AND RETURNS IT.

- THE PROBLEM WITH TASKS Here AND There IS THAT ONCE ONE OF THESE TASKS OBTAINS A BUFFER, THERE IS NO GUARANTEE THAT THE TASK WILL BE ABLE TO FINISH WITH THE BUFFER AND RETURN IT TO THE POOL. A TASK CAN RETURN ALL OF THE BUFFERS IT HAS ACQUIRED ONLY AFTER IT HAS OBTAINED ALL OF THE BUFFERS IT NEEDS.

- SIMILARLY, WE KNOW THAT A TASK CAN CLOSE File_1 AND File_2 (WHICH RETURNS THEM TO THE FILE SYSTEM) ONLY AFTER IT HAS SUCCESSFULLY OPENED BOTH FILES.

INSTRUCTOR NOTES

- THIS SLIDE SHOWS THE TWO POSSIBLE SOLUTIONS TO THE BUFFER ALLOCATION PROBLEM AND ONE POSSIBLE SOLUTION TO THE FILE SHARING PROBLEM.

- IN THE FIRST SOLUTION, There MAKES A REQUEST FOR THE FIRST BUFFER. WHEN IT GETS THE BUFFER, IT MAKES A REQUEST FOR THE SECOND BUFFER. IF IT DOES NOT GET THE SECOND BUFFER WITHIN ONE SECOND, IT RELEASES THE FIRST BUFFER AND STARTS ITS REQUESTS AGAIN. THIS ALLOWS Here TO GET ITS THIRD BUFFER, FINISH WITH THE BUFFERS, AND RETURN THEM TO THE POOL. There NOW HAS THE CHANCE TO GET ITS BUFFERS.

- TWO THINGS TO POINT OUT TO THE CLASS:

  1. IT IS STILL POSSIBLE THAT STARVATION MIGHT OCCUR. IF THE RUNTIME SYSTEM IS NOT FAIR, THEN TASK Here MIGHT GET ITS BUFFERS BACK BEFORE There CAN GET ITS BUFFERS.

  2. USING THIS TECHNIQUE REQUIRES CAREFUL CHOICE OF THE DELAY EXPRESSION. TOO LONG OF A DELAY MAY RESULT IN A SLUGGISH SYSTEM, WHILE TOO SMALL OF A DELAY MIGHT RESULT IN A SYSTEM THAT IS JUST SPINNING ITS WHEELS.

- IN THE SECOND SOLUTION, THE BUFFERS ALLOCATED ARE RETURNED IN A LIST. Buffer_List_Type IS AN UNCONSTRAINED ARRAY WHOSE COMPONENTS ARE BUFFER NUMBER. IF List_Of_Buffers_Allocated'Length = 0 THEN THE BUFFERS WERE NOT ALLOCATED. IF THE BUFFERS COULD NOT BE ALLOCATED, THEN A LOOP WITH A DELAY EXPRESSION COULD BE USED. MORE LIKELY, RATHER THAN USING LENGTH ATTRIBUTE, A SCHEDULER WOULD BE USED. (WE WILL TALK ABOUT SCHEDULING IN PART VI).

- NOTE THAT THE SECOND SOLUTION IS NOT FEASIBLE FOR THE FILE SHARING PROBLEM SINCE MOST SYSTEMS DO NOT ALLOW MULTIPLE FILES TO BE ACQUIRED SIMULTANEOUSLY.

HOW DO WE DEAL WITH THIS?

- IN THE BUFFER ALLOCATION EXAMPLE, WE NEED TO ALLOW THE TASKS TO ALLOCATE THEIR BUFFERS "ALL AT ONCE." TASK There COULD BE WRITTEN AS:

```
task body There is
   ...
begin
   loop
      Buffer_Allocation.Request (T_1);
      select
         Buffer_Allocation.Request (T_2);
         exit;   -- both buffers obtained
      or
         delay 1.0;                         -- IN CASE SOME OTHER TASK IS BLOCKED
         Buffer_Allocation.Release (T_1); -- GIVE UP BUFFER.
      end select;
   end loop;
   -- IF THIS POINT IS REACHED, THEN There WILL EVENTUALLY RETURN
   -- THE TWO BUFFERS TO THE POOL.
      ...
end There;
```

- ANOTHER APPROACH IS TO REWRITE REQUEST AND RELEASE TO ALLOW ALL BUFFERS NEEDS TO BE REQUESTED AT ONCE.

```
entry Request (Number_Of_Buffers_Requested : in Positive;
               List_Of_Buffers_Allocated   : out Buffer_List_Type);

entry Release (List_Of_Buffers_Allocated : in Buffer_List_Type);
```

- FOR THE FILE SHARING PROBLEM, ONLY THE FIRST SOLUTION IS REASONABLE.

INSTRUCTOR NOTES

- BULLET 1 - WE ASSUME NO TASK EVER ASKS FOR MORE OF THE RESOURCE THAN MAY EVER EXIST.

- BULLET 2 - THIS BULLET SHOWS WHY WE WANT TO TALK ABOUT AN AGENT TASK. THE AGENT TASK IS ACTING AS A MONITOR (MONITORS WILL BE MORE FULLY DISCUSSED LATER IN THE COURSE). WE WANT TO PUT THE EMPHASIS ON THE TASKS SHARING THE RESOURCES. NOT THE AGENT TASK (IN WHICH THE INTERACTION IS EXPLICIT).

DEADLOCK DUE TO RESOURCE SHARING

- TASKS THAT SHARE A FIXED SIZE POOL OF SOME PERMANENT RESOURCE NORMALLY DO SO THROUGH THE USE OF AN AGENT TASK.
  - WHEN A TASK WANTS TO OBTAIN SOME ITEMS OF THE RESOURCE, IT GETS THEM THROUGH THE AGENT TASK.
  - WHEN THE RESOURCES ARE TO BE RETURNED TO THE POOL, THEY ARE RETURNED THROUGH THE AGENT TASK.

- WHEN A TASK ASKS THE AGENT FOR SOME OF THE RESOURCE AND THERE IS NOT ENOUGH OF THE RESOURCE AVAILABLE, THE ASKING TASK IS BLOCKED. IF ENOUGH OF THE RESOURCE BECOMES AVAILABLE (BECAUSE OTHER TASKS HAVE RETURNED THEIR SHARE OF THE RESOURCE) A BLOCKED TASK MAY THEN BE GIVEN THE RESOURCES IT ASKED FOR, AND ALLOWED TO PROCEED. WE SAY THAT THE TASK IS UNBLOCKED.

- DEADLOCK OCCURS WITH TASKS SHARING A RESOURCE WHEN
  - ONE OR MORE TASKS ARE BLOCKED ON SOME RESOURCES
  - THE AGENT CAN NEVER SATISFY ALL REQUESTS, I.E., SOME OF THE TASKS WILL NEVER BE UNBLOCKED.

INSTRUCTOR NOTES

NECESSARY CONDITIONS FOR DEADLOCK IN RESOURCE SHARING

- THE FOLLOWING FOUR CONDITIONS ARE NECESSARY FOR DEADLOCK TO OCCUR.

  1. MUTUAL EXCLUSION - A RESOURCE ITEM CAN BE ACQUIRED BY ONLY ONE TASK AT A TIME.
  2. PARTIAL ALLOCATION - A TASK CAN ACQUIRE ITS RESOURCES PIECEMEAL.
  3. NON-SEIZURE - A RESOURCE ITEM CAN ONLY BE RELEASED BY THE TASK THAT ACQUIRED IT.
  4. CIRCULAR WAITING - TASKS ACQUIRE PART OF THEIR RESOURCES AND THEN WAIT TO ACQUIRE RESOURCES ACQUIRED BY OTHER TASKS.

- THE CONDITIONS ARE NECESSARY BUT NOT SUFFICIENT. (DEADLOCK NEED NOT OCCUR IF THE ABOVE CONDITIONS EXIST, BUT CANNOT OCCUR IF THE CONDITIONS DO NOT EXIST.)

```
   task body Here is                          task body There is
      ...                                        ...
   begin                                      begin
      Buffer_Allocation.Request (H_1);              ...
         ...                                     Buffer_Allocation.Request (T_1);
      Buffer_Allocation.Request (H_2);              ...
         ...                                     Buffer_Allocation.Request (T_2); ⬅
➡     Buffer_Allocation.Request (H_3);
         ...                                        ...
      Buffer_Allocation.Release (H_3);
         ...
      Buffer_Allocation.Release (H_2);              ...
         ...                                     Buffer_Allocation.Release (T_2);
      Buffer_Allocation.Release (H_1);              ...
         ...                                     Buffer_Allocation.Release (T_1);
   end Here;                                  end There;
```

- IN THIS EXAMPLE

  1. BUFFERS CAN BE USED BY ONLY ONE TASK AT A TIME
  2. EACH TASK GETS ITS BUFFERS ONE AT A TIME
  3. NEITHER TASK CAN SEIZE THE OTHER'S BUFFERS
  4. EACH TASK IS WAITING FOR THE OTHER TO GIVE UP ITS BUFFERS

INSTRUCTOR NOTES

- ITEM 1 - THE NEXT SEVERAL SLIDES TALK ABOUT AVOIDANCE. POINT OUT THAT THE MONITORING OF RESOURCES APPROACH REQUIRES SOME TYPE OF SCHEDULER. WE WILL TALK ABOUT THIS IN CONJUNCTION WITH CIRCULAR WAITING.

- ıTEM 2 - THIS IS ABOUT ALL WE ARE GOING TO SAY ABOUT RECOVERY. TELL THE CLASS THAT THIS IS DISCUSSED IN THE REFERENCES -ESPECIALLY Shaw AND Holt - BUT IS BEYOND THE SCOPE OF THE COURSE.

POLICIES FOR DEALING WITH DEADLOCK

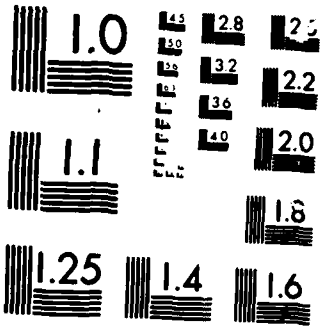
- POLICIES FOR DEALING WITH DEADLOCK ARE:
  - AVOID IT - NEVER LET THE ABOVE CONDITIONS OCCUR.
    - REQUIRES DISCIPLINED TASK INTERACTION (DESIGN DECISION), OR
    - REQUIRES MONITORING OF RESOURCE USAGE (EXECUTION TIME DECISION)
  - RECOVER FROM IT - WHEN DEADLOCK IS DETECTED, DEAL WITH IT.
    - ABORT ONE OR MORE OF THE DEADLOCKED TASKS UNTIL ENOUGH RESOURCES ARE FREED UP. (OKAY FOR COMMERCIAL OPERATING SYSTEMS, BUT NOT FOR AN AVIONICS PROGRAM).
    - RESOURCES CAN BE SEIZED TO ALLOW ONE OR MORE OF THE DEADLOCKED TASKS TO PROCEED. THE RESOURCES ARE EVENTUALLY RETURNED TO THE TASK THEY WERE SEIZED FROM.

INSTRUCTOR NOTES

- BULLET 1 - THE ASSUMPTION MEANS THAT THE TASK DOES NOT CONTAIN STATEMENTS THAT WILL NEVER FINISH SUCH AS INFINITE LOOPS OR EXPLICIT INTERACTIONS WITH TASKS THAT RESULT IN DEADLOCK.

AD-A166 351 ADA (TRADE NAME) TRAINING CURRICULUM REAL-TIME SYSTEMS IN ADA L401 TEACHER'S GUIDE VOLUME 1(U) SOFTECH INC WALTHAM MA 1986 DAAB07-83-C-K514 5/5

UNCLASSIFIED F/G 5/9 NL



MICROCOPY RESOLUTION TEST CHART

AVOIDING DEADLOCK IN RESOURCE SHARING

- SINCE DEADLOCK CAN ONLY OCCUR WHEN ALL FOUR CONDITIONS HOLD, DEADLOCK CAN BE PREVENTED BY ENSURING THAT NO MORE THAN THREE OF THE CONDITIONS HOLD.

- AVOIDING MUTUAL EXCLUSION:
  - PERMIT SEVERAL TASKS TO ACCESS RESOURCES SIMULTANEOUSLY
  - THIS COULD WORK FOR TASKS HAVING READ-ONLY ACCESS TO COMMON DATA.
  - TASKS SOMETIMES NEED EXCLUSIVE ACCESS TO RESOURCES, SO THIS APPROACH IS NOT ALWAYS FEASIBLE.

- AVOIDING PARTIAL ALLOCATION OF RESOURCES:
  - TASKS MUST KNOW THEIR MAXIMUM RESOURCE NEEDS AND REQUEST THEM ALL AT ONCE. THEY CANNOT PROCEED UNTIL ALL NEEDED RESOURCES ARE ALLOCATED TO THEM.
  - THIS CAN BE COSTLY, SINCE RESOURCES THAT WON'T BE NEEDED FOR A WHILE MUST BE ALLOCATED IN ADVANCE. (OF COURSE, A TASK SHOULD RETURN ANY RESOURCE AS SOON AS IT IS DONE WITH IT).

INSTRUCTOR NOTES

- BULLET 1
  - ITEM 3 - THE COST CAN BE IN DECREASED EFFICIENCY IN THE UTILIZATION OF RESOURCES. IF CARE IS NOT TAKEN, PREEMPTION COULD "EAT UP" A GREAT DEAL OF PROCESSOR TIME. HOWEVER, A REASONABLE AMOUNT OF DECREASED EFFICIENCY MAY BE ACCEPTABLE.
  - ITEM 4 - THIS IS GENERALLY NOT REASONABLE FOR FILE SHARING UNLESS THE FIRST APPROACH IN ITEM 1 IS USED, AND THE TASK KNOWS WHAT FILES IT WILL NEED. OTHERWISE, FORCIBLE PREEMPTION WOULD LEAD TO FILES IN INCONSISTENT STATES.

AVOIDING DEADLOCK IN RESOURCE SHARING (CONTINUED)

- ALLOWING PREEMPTIVE SEIZURES: NON-PREEMPTIVE SCHEDULING
  - FIRST APPROACH - WHEN A TASK HOLDING SOME RESOURCES IS DENIED ADDITIONAL RESOURCES, IT SHOULD RELEASE ITS ORIGINAL RESOURCES AND REQUEST THE ORIGINAL AND ADDITIONAL RESOURCES AGAIN. (SELF-PREEMPTION).
  - SECOND APPROACH - A TASK'S RESOURCES ARE TAKEN FROM IT AND GIVEN TO ANOTHER TASK. EVENTUALLY THE TASK IS GIVEN BACK ITS RESOURCES. (PREEMPTIVE SCHEDULING).
  - COST OF PREEMPTION CAN BE HIGH.
  - PRACTICAL ONLY FOR RESOURCES WHOSE STATES CAN BE SAVED AND RESTORED.
    - PRACTICAL FOR PROCESSORS, MAIN MEMORY.
    - NOT PRACTICAL FOR PRINTERS, TAPE UNITS.

INSTRUCTOR NOTES

- USING THE EXAMPLE IN ITEM 2, EXPLAIN WHY DEADLOCK CANNOT OCCUR.
  - SUPPOSE A TASK HAS ACQUIRED THE CONTROL PANEL. SINCE THERE IS NO HIGHER RESOURCE, THE TASK MUST EVENTUALLY RELEASE THE PANEL ONCE IT GETS IT. SINCE PARTIAL ALLOCATIONS ARE NOT ALLOWED WITHIN A CLASS, TASKS CANNOT DEADLOCK WITH RESPECT TO THE CONTROL PANEL.
  - NOW SUPPOSE THAT A TASK HAS ACQUIRED MAIN MEMORY. THE ONLY WAY IT CAN FAIL TO RELEASE MAIN MEMORY IS IF IT HANGS UP WAITING FOR THE CONTROL PANEL. BUT WE JUST SAW THAT THIS CANNOT HAPPEN, SO THE TASK MUST EVENTUALLY FREE UP MAIN MEMORY. SIMILARLY FOR THE DISPLAY.

- THE ABOVE "EXPLANATION" FOLLOWS THE PATTERN OF THE INDUCTIVE PROOF GIVEN BY BRINCH HANSEN IN OPERATING SYSTEM PRINCIPLES.

AVOIDING DEADLOCK IN RESOURCE SHARING (CONTINUED)

- AVOID CIRCULAR WAITING: HIERARCHIC ORDERING OF RESOURCES PROVIDES A SECOND APPROACH.
  - RESOURCES ARE PARTITIONED INTO CLASSES WITH EACH CLASS DEFINING A LEVEL IN THE HIERARCHY. (EACH CLASS MAY CONTAIN MORE THAN ONE TYPE OF RESOURCE).
  - EXAMPLE: LEVEL 1 - DISPLAY; LEVEL 2 - MAIN MEMORY; LEVEL 3 - CONTROL PANEL.
  - A TASK ALLOCATES RESOURCES IN A CLASS VIA A SINGLE REQUEST. (CAN REQUEST 2 BLOCKS OF MEMORY, NOT 1 BLOCK TWICE).
  - ONCE A TASK ALLOCATES RESOURCES FROM A CLASS AT A GIVEN LEVEL, IT CAN ONLY REQUEST RESOURCES FROM HIGHER LEVELS. (ONCE MAIN MEMORY IS ACQUIRED, ONLY THE CONTROL PANEL MAY BE REQUESTED).
  - A TASK MAY RELEASE RESOURCES FROM A CLASS AT A GIVEN LEVEL, ONLY IF IT IS NOT HOLDING ANY RESOURCES AT A HIGHER LEVEL. (A TASK MUST RELEASE MAIN MEMORY BEFORE IT CAN RELEASE THE DISPLAY).
  - WHEN A TASK RELEASES RESOURCES FROM A CLASS, IT CAN AGAIN REQUEST RESOURCES FROM THE CLASS. (A TASK MAY RELEASE ITS BLOCKS OF MEMORY AND THEN ASK FOR SOME BACK AGAIN).
  - ASSUME THAT A TASK WILL EVENTUALLY RELEASE ALL OF ITS RESOURCES AT A GIVEN LEVEL, UNLESS DELAYED INDEFINITELY BY REQUESTS AT HIGHER LEVELS.
  - THESE ASSUMPTIONS LET US CONCLUDE THAT DEADLOCK CANNOT OCCUR.

Material: Real Time Systems in Ada (L401), Volume I A146782

We would appreciate your comments on this material and would like you to complete this brief questionaire. The completed questionaire should be forwarded to the address on the back of this page. Thank you in advance for your time and effort.

1. Your name, company or affiliation, address and phone number.

2. Was the material accurate and technically correct?

Yes □ No □

Comments:

3. Were there any typographical errors?

Yes □ No □

If yes, on what pages?

4. Was the material organized and presented appropriately for your applications?

Yes □ No □

Comments:

5. General Comments:

place
stamp
here

COMMANDER
US ARMY MATERIEL COMMAND
ATTN: AMCDE-SB (OGLESBY)
5001 EISENHOWER AVENUE
ALEXANDRIA, VIRGINIA 22233